VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Optimalizace pomocí roje částic

Particle Swarm Optimization

2020 Bc. Vít Doleží

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Vít Doleží**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Optimalizace pomocí roje částic
Particle Swarm Optimization

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je implementace vybraného algoritmu na GPU, provedení výkonnostních experimentů navržené implementace, zhodnocení výsledků, stanovení limitů pro výpočty a závěrečná sumarizace těchto testů. Následně bude implementace začleněna do ukázkové interaktivní aplikace pro vizualizaci průběhu výpočtu.

Hlavní body zadání.
1. Seznámení se s problematikou paralelizace algoritmů na GPU.
2. Návrh a implementace vybraného bio-inspirovaného algoritmu na GPU.
3. Test navrženého algoritmu nad konkrétní úlohou hledání optimálního řešení úlohy, vyhodnocení testů.
4. Integrace navrženého řešení do ukázkové aplikace pro vizualizaci.
5. Shrnutí dasažených výsledků.

Seznam doporučené odborné literatury:

[1] Edward Angel: Interactive Computer Graphics, ISBN-10:032153586 (2009)
[2] Jason Standers, Edward Kandrot: CUDA by Example: An Introduction to General-Purpose GPU Programming, ISBN-10: 0131387685, 2010

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **doc. Ing. Petr Gajdoš, Ph.D.**

Datum zadání: 01.09.2019
Datum odevzdání: 30.04.2020

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 15. 5. 2020 .....

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.

V Ostravě 15. 5. 2020 .....

Rád bych poděkoval doc. Ing. Petrovi Gajdošovi, Ph.D. za odbornou pomoc a konzultaci při vytváření této diplomové práce.

Abstrakt

Tato diplomová práce se zabývá aplikací Evolučních algoritmů na optimalizační úlohy a paralelizaci těchto algoritmů na GPU. Hlavním cílem práce bylo naimplementovat evoluční algoritmus Optimalizace pomocí roje částic na grafické kartě pomocí CUDA API, provést výkonnostní testy tohoto algoritmu, a také použít implementaci v ukázkové aplikaci, kde se vizualizují částice tohoto algoritmu. Pro porovnání byly implementovány navíc algoritmy diferenciální evoluce a optimalizace mravenčí kolonii, všechny algoritmy byly implementovány v technologii CUDA, v jayzce C++ s OpenMP a v jazyce Python. V nejlepších případech bylo zrychlení na grafické kartě až 15200násobné, v nejhorším případě pouze 86× rychlejší než Python implementace.

Klíčová slova: Evoluční algoritmy, Optimalizace pomocí roje částic, Diferenciální evoluce, Optimalizace mravenčí kolonii, paralelní výpočty, CUDA, Python, C++, OpenMP, OpenGL

Abstract

This master's thesis deals with application of Evolutionary algorithms on optimization problems and parallelization of those algorithms on GPU. The aim of this work was to implement Particle swarm optimization on a graphics card with CUDA API, then make performance benchmarks to test it and showcase it in the application, where particles of this algorithm are visualized. For better comparison, algorithms like Differential evolution and Ant colony optimization were implemented too, all used algorithms were implemented in CUDA, C++ with OpenMP and Python. In the best-case scenario, the GPU implementation was 15200 times faster, in the worst-case, the speedup was only 86 times faster than Python.

Keywords: Evolution algorithms, Particle swarm optimization, Differential evolution, Ant colony optimization, parallel computing, CUDA, Python, C++, OpenMP, OpenGL

Obsah

Seznam použitých zkratek a symbolů

EA	–	Evoluční Algoritmus
PSO	–	Particle swarm optimization
DE	–	Diferenciální evoluce
ACO	–	Ant colony optimization
SISD	–	Single instruction stream, single data stream
SIMD	–	Single instruction stream, multiple data streams
MISD	–	Multiple instruction streams, single data stream
MIMD	–	Multiple instruction streams, multiple data streams
SIMT	–	Single instruction multiple threads
ALU	–	Arithmetic logic unit
FPU	–	Floating point unit

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Evoluční algoritmy jsou určeny pro problémy, pro které není možné nalézt řešení polynomiálním deterministickým algoritmem. Jsou použity u problémů, které mají velké množství vstupních parametrů, kde je potřeba zjistit kombinaci těchto parametrů, která dává nejlepší výsledek. Tyto algoritmy májí nějakou populaci, kde se každý jedinec chová podle předem určených pravidel. Tito jedinci většinou nejsou na sobě závislí, každý jedinec může svůj proces vykonat nezávisle na druhém. Dále tyto algoritmy probíhají v iteracích, kdy se opakovaně za sebou zkouší najít nové řešení. Toto předpoklady jsou ideální pro paralelizaci na grafických kartách, která může tisíce těchto jedinců vyzkoušet v jeden čas, oproti typickému procesoru, který může najednou vyzkoušet pouze 4 - 8 jedinců. V posledních letech se výkony grafických karet zvyšují hodně rychleji než výkony procesorů. Karty jsou využívaný ve více odvětvích, jako jsou třeba neuronové sítě nebo zpracování obrazu.

Text je rozdělený na 4 části, v první části je seznámení se s evolučními algoritmy a paralelním programováním. Na úvodu je termín evoluční algoritmus porovnán, jak jej vnímají v jiných literaturách, dále jsou popsány vybrané algoritmy, zmíní se rozdíly mezi verzemi jednotlivých algoritmů. Je zde taky popsán, jak by měl vypadat obecný evoluční algoritmus, a že ne všechny evoluční algoritmy samy každý z těchto procesů využívají.

Druhá část se zabývá paralelním programování. Jsou zde popsány základní paralelní architektury, je zde zmíněna technologie OpenMP pro paralelizaci na procesoru a nakonec je zde popsána technologie CUDA, v čem se liší od normálního programování, jak vypadá typický běh CUDA programu a další věci ohledně programování na grafické kartě.

V třetí části jsou více popsány algoritmy, které byly skutečně implementovány. Jsou to optimalizace pomocí roje částic, diferenciální evoluce a optimalizace mravenčí kolonii. Implementovaný algoritmus je zde vždy popsán pomocí vzorců nebo pseudokodu a jsou zde zmíněny problémy, které při paralelní implementaci mohly nastat. Dále je zde popsán způsob hledání nejlepšího jedince na grafické kartě pomocí paralelního skenu, jak vlastně pracuje a jeho optimalizace. Nakonec je zde popsána vizualizace roje částic, kde se používá a na jakých datech tento algoritmus hledá globální minimum

Ve čtvrté části jsou pak implementované algoritmy otestované. Je zde seznam matematických funkcí, na kterých bylo PSO a DE otestováno a také datasety, na kterých bylo otestováno ACO na problému obchodního cestujícího. Dále je zde porovnána Python a CUDA implementace a jak velký přínos implementace na grafické kartě měla. Na funkcích je otestována PSO proti DE, kde se zjišťuje, který algoritmus má za stejný čas lepší výsledek. Nakonec je zde otestována vizualizace a jak velký počet částic zvládne zobrazit.

2 Evoluční algoritmy

V této kapitole budou popsány základní pojmy a terminologie Evolučních algoritmů a co pod pojmem EA někdy myslí ostatní autoři, budou popsány její různé druhy, a čím se mezi sebou liší. Bude taky popsaná obecná implementace takové algoritmu.

Evolučním algoritmy jsou inspirovány chováním v přírodě, jejich účelem je najít přibližné řešení pro daný problém. EA většinou pracují s nějakou populací, kde se každý jedinec populace chová podle stejných pravidel. V EA je pět hlavních procesů, inicializace populace, selekce, křížení, mutace a ukončení. Základní myšlenka je v těchto algoritmech stejná, silní jedinci přežijí, a udělají nové potomky, zatímco slabí jedinci umřou a jejich geny nebudou v dalších generacích.

2.1 Terminologie evolučních algoritmů

Někteří autoři používají termín evoluční výpočetní techniky při popisu EA [8] Tímto zdůrazňují, že EA jsou implementovány na počítačích. Nicméně evoluční výpočetní techniky mohou odkazovat na algoritmy, které nejsou používány pro optimalizaci, například první genetické algoritmy nebyly přímo používány pro optimalizaci, ale byly zamýšleny pro studium přirozeného výběru [9]. Algoritmy v této práci jsou zaměřené na evoluční optimalizační algoritmy, což je víc specifický pojem než evoluční výpočetní techniky.

Soft computing je další termín související s EA [10]. Hard computing znamená přesné, precizní, numerické výpočty. Soft computing poukazuje na méně přesné výpočty. Soft computing algoritmy vypočítají dobré výsledky pro problémy, které jsou například složité, zašumělé, více modální nebo více objektivní. A tedy EA jsou podmnožinou Soft computing.

Další termíny jsou přírodou inspirované výpočty nebo bio inspirované výpočty [11]. Nicméně, některé EA, jako diferenciální evoluce nebo odhady distribučních algoritmů nemusí být nutně motivovány přírodou. Další algoritmy, jako jsou evoluční strategie, mají jen málo společné s procesy v přírodě. EA jsou více obecné než přírodou inspirované výpočty, protože zahrnují i algoritmy, co nejsou úplně inspirované přírodou [12].

Další autoři používají pojem optimalizace založena na populaci [9]. Toto klade důraz na to, že EA se všeobecně skládá z populace kandidátních řešení. Jedinci této populace se v průběhu čas vyvíjí k lepšímu řešení daného problému. Některé EA ale mohou mít pouze jednoho kandidáta na řešení v každé jejich iteraci, například Hill climbing nebo evoluční strategie. EA jsou více obecné než optimalizace založené na populaci, mají pod sebou i algoritmy z jedním kandidátním řešením.

Další hodně používaný termín je strojové učení [13]. Jsou to počítačové algoritmy, které se učí ze zkušeností. Tento obor zahrnuje hodně dalších algoritmů než je pouze EA. Strojové učení je obecně bráno jako širší obor než EA, zahrnuje v sobě obory jako neuronové sítě, podpůrné učení, shlukování, support vector machines a další.

Někteří autoři používají termín heuristické algoritmy [14]. Ty algoritmy používají metody přístupu *common sense* nebo *rule of thumb* pro vyřešení problému. Od heuristických algoritmů

se neočekává, že najdou nejlepší řešení pro daný problém, ale čeká se, že najdou dostatečné dobré řešení problému. Termín metaheuristické je pojem pro část heuristických algoritmů. EA, které lze implementovat více způsoby s více nastavením a parametry jsou metaheuristické [15].

Terminologie je nepřesná a v publikacích hodně záleží na kontextu. V této práci se evoluční algoritmus EA bere jako algoritmus, co hledá lepší a lepší řešení problému za každou iteraci. Jedna iterace EA se většinou nazývá generace, aby se zachoval odkaz na přírodu. Základní myšlenka je ale pro všechny tyto techniky stejná, vliv prostředí na populaci má zajistit přirozený výběr (*natural selection, survival of the fittest*), v důsledku toho se zvyšuje kvalita populace. EA používají mechanismy, které jsou inspirovány biologickou evolucí jako reprodukce, mutace, křížení a výběr [9]. Hlavním úkolem EA je najít co nejlepší řešení pro daný problém. Kvalita řešení je reprezentována hodnotou účelové funkce. Tuto hodnotu se EA snaží buď minimalizovat, nebo maximalizovat. Konečné řešení nemusí být nutně to nejlepší řešení, EA dává pouze aproximační řešení.

2.2 Obecná implementace evolučního algoritmu

Na začátku se musí stanovit parametry evoluce, každý algoritmus má své parametry, jako je třeba uzavření populace v nějakém rozsahu hodnot, počet iterací, velikost populace a parametry pro konkrétní algoritmus. Dále se definuje účelová funkce, ať je co optimalizovat a kritérium pro konec výpočtu, může to být nějaká prahová hodnota účelové funkce nebo počet iterací.

Vygeneruje se počáteční populace. Jedinec populace je vektor čísel, kde každé číslo představuje parametr účelové funkce. Počet parametrů účelové funkce je dimenze problému. Každé číslo se vygeneruje v povoleném rozsahu. Každý jedinec se ohodnotí podle dané účelové funkce. Hodnota této funkce bude rozhodovat o vhodnosti jedince pro vyřešení úlohy.

Na základě účelové hodnoty se vyberou ti nejlepší kandidáti jako základ pro další generaci (rodičové), která se vytvoří pomoci křížení a mutace. Křížení se aplikuje na dva nebo více rodičů a výsledkem jsou jeden nebo více nových kandidátů (potomků). Mutace se aplikuje na jednoho jedince a výsledkem je zase jeden jedinec. Tito nově vytvoření kandidáti se opět ohodnotí podle účelové funkce. Ti nejlepší jedinci z nových kandidátů a ze staré generace se vyberou jako jedinci do nové generace, zbytek jedinců se zahazuje. Tento proces je několikrát opakován, dokud není nalezeno dostatečně dobré řešení nebo nastane maximální počet opakování [16]. V algorithmu 1 je pseudokod pro obecný postup evolučního algoritmu.

2.3 Hill climbing

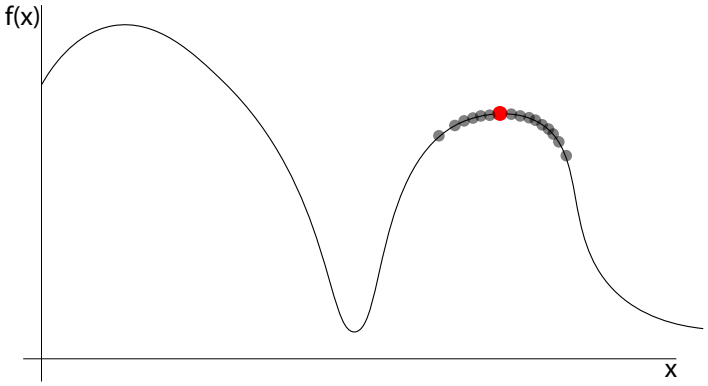
Hill climbing je jeden z nejjednodušších evolučních algoritmů. Jeho jednoduchost je dobrý důvod, proč při řešení optimalizačních problémů tento algoritmus zkusit jako první. Hlavní myšlenka základní verze tohoto algoritmu je, pokud je potřeba se dostat na nejvyšší bod plochy, z místa kde se jedinec nachází, se udělá krok na místo, které je v tomto lokálním okolí nejvyšší. Nové místo se vždy použije jako střed pro okolí dalšího kroku.

Algorithm 1 Pseudokod pro obecný Evoluční Algoritmus.

```
1: Inicializace populace
2: Ohodnocení populace
3: while Podmínka pro konec výpočtu do
4:     Výběr nejlepších rodičů
5:     Křížení rodičů
6:     Mutace potomků
7:     Výběr nejlepších kandidátů pro další generaci
8: end while
```

Na hledací ploše se vybere jeden náhodný bod. Kolem tohoto bodu se v okolí vygeneruje několik náhodných bodů. Všechny tyto body se ohodnotí účelovou funkcí a ten nejvhodnější z nich se vybere jako střed pro další okolí. Pokud žádný z těchto bodů nemá lepší ohodnocení, střed pro další okolí zůstává původní bod. Toto se opakuje, dokud nenastane podmínka pro zastavení, která je většinou zadaná předem jako maximální počet iterací.

Slabina tohoto algoritmu je, že se na více modálních funkcích může lehce zaseknout v lokálních extrémech. Znázorněno na obrázku 1, kde červený bod je střed okolí, a šedé body jsou kandidáti pro další krok, žádný z kandidátů nemá lepší ohodnocení než středový bod, ale z pohledu na funkci je jasné, červený bod není v globálním extrému. Toto se dá vyřešit buď zvětšení okolí pro generování dalších kroků, což by umožnilo udělat krok na vyšší místo na funkci, než kde se aktuální střed nachází, nebo celý algoritmus spouštět vícekrát. Algoritmus by tak začal na funkci pokaždé v jiném místě, a mohl by pokaždé najít jiný extrém. Z těchto několika výsledku se nakonec vybere ten nejlepší, zvyšuje se tak pravděpodobnost, že daný výsledek opravdu bude globální extrém. Tento přístup se nazývá random restart hill climbing [9].



Obrázek 1: Hill climbing - uvíznutí v lokálním extrému

2.4 Diferenciální evoluce

A jak už bylo řečeno, diferenciální evoluce (DE) není inspirována procesy v přírodě. Je to algoritmus založený na populaci, každý jedinec v populaci je vektor, kandidát na řešení problému. DE je založena na myšlence, pro každého jedince v populaci vzít dva jiné jedince, ty od sebe odečíst (diferenční vektor) a poté je vynásobit mutační konstantou (váhový diferenční vektor), tento výsledek nakonec přičíst ke třetímu vektoru z populace a vytvořit tak nové kandidátní řešení (šumový vektor), které je podle náhody zkombinováno s původním vybraným jedincem a vytvoří tak zkušební vektor [17].

Existuje mnoho různých typů DE, které například berou v potaz elitismus, nebo diferenční vektor počítají z více vektorů nebo jiným způsobem skládají zkušební vektor [18]. V této práce se používá verze *DE/rand/1/bin* a je ve větším detailu popsána v kapitole Implementace diferenciální evoluce.

2.5 Hejnové algoritmy

Hodně autorů odděluje EA od hejnových algoritmů. Inteligence skupiny můžeme pozorovat v mnoha systémech v přírodě. Dva z takových příkladů jsou kolonie mravenců nebo hejna ptáků. V takovém systému se dbá především na celkovou inteligenci skupiny a nehledí se příliš na inteligenci pouze jednoho jedince. Optimalizace mravenčí kolonií (*Ant colony optimization*) a optimalizace pomocí roje částic (*Particle swarm optimization*) jsou 2 hlavní hejnové algoritmy, a mnoho autorů tvrdí, že by se neměly klasifikovat jako EA, další zase tvrdí že by se hejnové algorimy měly považovat za součást EA [9]. Například jeden z autorů optimalizace pomocí roje částic jej považuje jako EA, protože algoritmus je vykonáván stejně jako obecný EA, viz vyvíjející se populace kandidátních řešení problému, která se vylepšuje v každé iteraci [19].

2.5.1 Optimalizace pomocí roje částic

Optimalizace pomocí roje částic (PSO) je založena na tom, že skupina jedinců, která pracuje na nějaké úloze, vylepšuje nejen inteligenci a výkon skupiny jako celku, ale taky se vylepšuje každý jedinec sám. V přírodě toto chování můžeme vidět například, když se hejna ptáků nebo ryb snaží vyhnout predátorovi. Velká skupina zvířat se může predátorovi jevit jako větší a hlasitější, než samostatný pták či ryba, a tímto se tak mohou vyhnout útoku. Další příklad může být hledání potravy. Principy PSO je také možno vidět v lidech. Používáme k řešení dovednosti, které se nám v minulosti osvědčili a víme, že fungují. Snažíme se napodobit jedince, kteří v něčem už úspěšní byli. Snažíme se naučit jejich taktiky z knih nebo internetu [20].

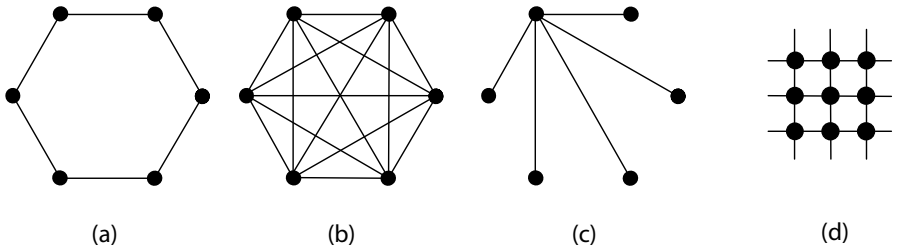
PSO je technika, která iterativně snaží optimalizovat své řešení. Problém řeší svou populací kandidátních řešení/jedinců, kteří se zde nazývají částice. Tyto částice se pohybují po vyhledávacím prostoru na základě matematických vzorců pro stanovení jejich pozice a rychlosti.

Částice mají definovanou svojí prvotní pozici a vektor rychlosti, který určuje jejich pohyb. Tento pohyb je s porovnáním k ostatním evolučním algoritmům u PSO originální, ostatní algo-

ritmy nic jako simulaci pohybu jednoho jedince neřeší, dělají spíše nového jedince, PSO ale zase nevyužívá křížení ani mutaci. Při pohybu se tyto rychlosti mění podle vzorců. Směr rychlosti a její velikost se mění hlavně na základě dvou faktorů. Jeden z nich je, že si částice pamatuje svou vlastní nejlepší pozici a snaží se k ní vrátit. Přirovnání v přírodě může být pták, co si pamatuje, kde se dobře najedl a chce se tam vrátit. Druhý faktor je nejlepší pozice nějakého jedince ze skupiny. Oba tyto faktory mají vliv na novou rychlost, vliv každé této složky na novou rychlost je dán náhodně, ale lze ho ovlivnit i parametry. Více informací o vzorcích a parametrech PSO bude popsáno v následující kapitole 4.1.

Počáteční rychlosti mohou být buď zvoleny náhodně, nebo mohou být nastaveny na nulu. Může se definovat velikost skupiny (počet sousedů), která bude mít vliv na rychlost částice. Tato velikost může být třeba dva, nebo můžeme brát skupinu jako celou populaci. Pro menší skupiny může být těžší uvíznout v lokálním extrému, větší skupiny zase rychleji konvergují k výsledku. Musí se limitovat maximální rychlost, aby částice nevyletěly z hledaného prostoru, který je většinou omezen předem danými hranicemi. To i při limitování maximální rychlosti může nastat, proto se po každém pohnutí částice kontroluje její pozice, a pokud je mimo hranice, tak se nastaví, aby stála na hranici. Může se taky použít elitismus, skupina si tak bude pamatovat nejlepší pozici i při novém kroku, ta se pak může použít místo nejlepší aktuální pozice sousedů.

Pohyb částice je závislý na nejlepší pozici sousedních částic, způsob uspořádání částic a výběr částic, co do skupiny patří, se nazývá topologie. V základním PSO se skupina sousedů vytvoří z několika nejbližších částic. Když se částice pohne, tak noví sousedé můžou být v nové migraci úplně jiní než ti ve staré, tato topologie se jmenuje dynamická. Skupiny se taky mohou definovat na začátku algoritmu, pak se používá označení statická topologie. Podle počtu sousedů se pak topologie značí na *gBest*, pokud k ve skupině všechny částice z populace, a *lBest*, pokud jsou skupiny menší než celá populace.



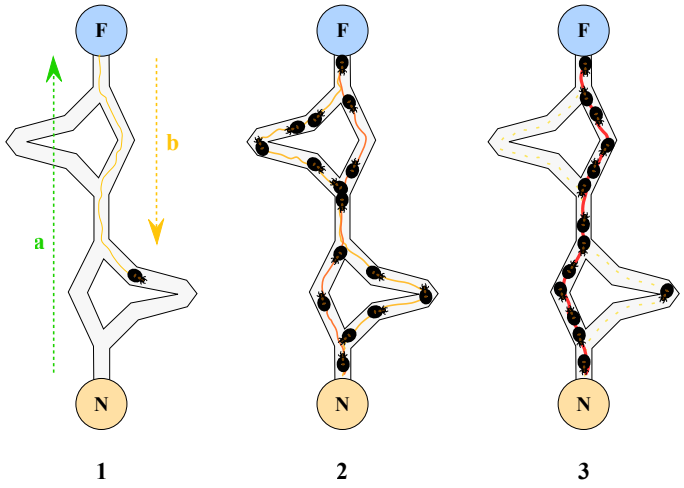
Obrázek 2: PSO topologie, (a) ring, (b) cluster, (c) wheel, (d) von Neumann

Další používané topologie, jsou ring, kde má každá částice právě 2 sousedy. Cluster, kde je každá částice propojena se všemi sousedy, některé tyto částice mohou být dále propojené s jinými clustery. Topologie wheel, kde jedna částice se nazývá *focal*, ta je propojena se všemi částicemi ve skupině, ale ostatní jsou napojeny pouze na hlavní *focal* částici. Dále topologie von Neumann, kde každá částice má 4 sousedy, na obrázku 2 krajní částice jsou propojeny s částicemi na opačné straně [9].

2.5.2 Optimalizace mravenčí kolonií

Optimalizace mravenčí kolonií (ACO) je inspirována z reálného světa z kolonií mravenců. Nejčastěji se požívá při hledání nejkratší cesty v grafu. Mravenci komunikují hlavně pomocí vypouštění feromonů. Když mravenci cestují z mraveniště ven hledat potravu a najdou ji, tak při cestě s úlovkem zpátky do mraveniště za sebou nechávají stopu s feromony. Tyto feromony vycítí další mravenci, pro které je to signál, že tímto směrem se nachází jídlo. Když na konci jídlo najdou i další, tak ho zase zanesou zpátky do mraveniště, při čemž taky vypouštějí feromony. Tímto se feromonová stopa ještě víc zesílí a přiláká ještě více dalších mravenců. Postupem času, cesta s nejvyšší sílou feromonů bude ta nejkratší cesta k jídlu [21].

Někdy mezi mraveništěm a jídlem může vzniknout překážka, což může zapříčinit, že cesta co na sobě má nejvíce feromonů, již nepovede k jídlu. Mravenci tak začnou hledat jiné cesty. Pokud se mravenec vrací do mraveniště bez jídla, tak žádné feromony nevylučuje. Časem se feromony z původní cesty, která vedla k jídlu, vypaří a mravenci ji budou používat méně. Budou se snažit více využívat jiné cesty.



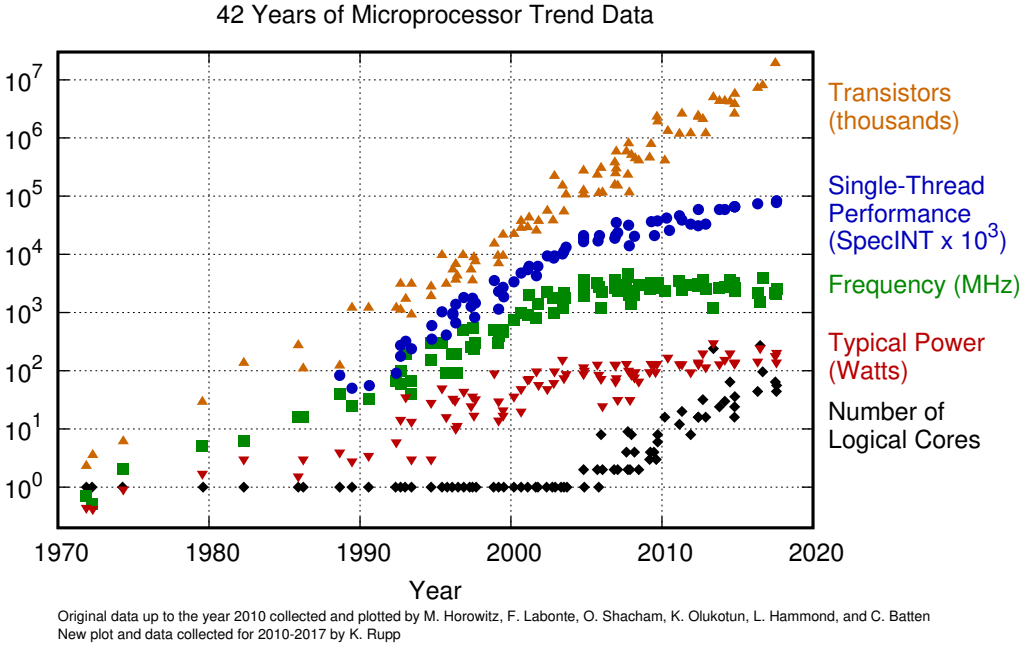
Obrázek 3: Graf z mraveniště k jídlu [1]

Pohyb mravenců je znázorněn na obrázku 3. Mravenec vyráží z mraveniště N pro jídlo F. Při cestě zpět vypouští feromony. Síla feromonů je zatím mála a tak další mravenci zkouší i jiné cesty k jídlu. Mravenci, kteří si vybrali tu nejkratší cestu se vrátí do mraveniště rychleji, než ti, co si vybrali delší cestu. To má za následek, že kratší cestu budou mravenci navštěvovat častěji, takže zde bude více feromonů. Mravenci používají nejsilnější cestu s feromony a při cestě zpět ji opět zesílí dalšími feromony. Někteří mravenci si i na posledním obrázku vybrali jinou než optimální cestu, při rozhodování totiž nezávisí pouze na feromonech, ale do procesu je také přidaná náhoda. V této práci je implementovaná základní ACO na problému obchodního cestujícího *Travelling salesman problem*. Existují i další verze jako Ant Colony System, Max-Min Ant System, Elitist Ant System viz [22].

3 Technologie pro paralelní programování

Většina programů, které programátoři píšou, jsou sériové programy. Sériový program běží na jednom procesoru v jednom vlákně. Instrukce v tomto programu jsou vykonávány za sebou v sérii, a pouze jedna instrukce je vykonána v jednom čase. Paralelní programování umožňuje více instrukcí v programu provádět zároveň, paralelně. Aby toto bylo možné, program musí být rozdělen do částí, které jsou na sobě nezávislé, aby každý procesor nebo vlákno, mohl vykonávat program zároveň s ostatními procesory. Paralelní výpočet může být dosažen na jednom počítači s více procesory nebo na více počítačích zapojených do sítě.

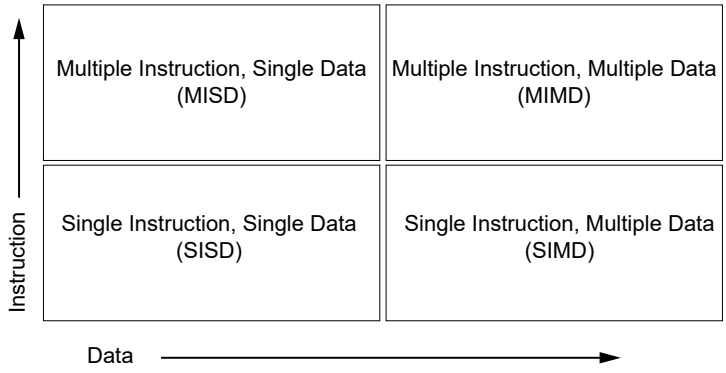
S příchodem více jádrových procesorů a obrovským výkonem grafických karet v nedávné době se paralelismus stal jedním z nejefektivnějších způsobů jak využít počítačový výkon. Na základě dat z grafu 4 lze vyčíst, že v zhruba v roce 2004 nastal zlom, a vývoj výkonu jednoho jádra v procesoru se značně zpomalil. Pokud byl před tímto rokem problém s výkonosti nějakého programu, stačilo počkat jeden až dva roky a pravděpodobně pak přišel nový procesor, který rychlost programu značně zvýšil a problém tedy vyřešil. Po tomto roce se výrobci procesorů snažili vydat jiným směrem, než je čisté zvyšování taktu jednoho jádra a začali přidávat na procesory více jader. Je jenom otázkou času, kdy ale znovu narazíme na fyzikální limity při zmenšování tranzistorů, a více jader na čip už taky nebude možné přidat.



Obrázek 4: Historie parametrů procesorů [2]

3.1 Architektury paralelního programování

Flynnova taxonomie se používá pro klasifikaci architektury počítače, znázorněno na obrázku 5. Klasifikuje systém na základě kolik instrukčních streamů a kolik datových proudů může architektura zpracovat najednou. Základní von Neummanova architektura počítače je tedy single instruction, single data SISD systém. Může zpracovávat pouze jednu instrukci nad jednou jednotkou dat v jeden čas. MISD systémy mohou vykonávat více instrukcí nad jedním datovým streamem, takto architektura je netradiční, příklad použití je kontrolní počítač letu raketoplánu [23].



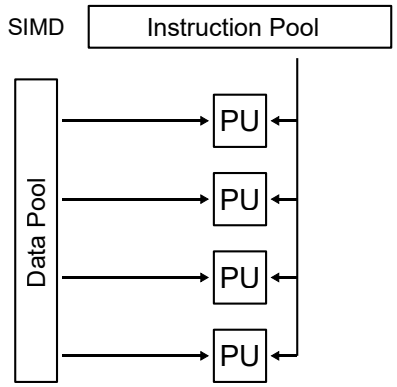
Obrázek 5: Flynnova taxonomie počítačové architektury

3.1.1 SIMD

SIMD systémy operují tak, že aplikují jednu stejnou instrukci na více jednotek dat. Abstraktně může být systém vysvětlen jako jedna kontrolní jednotka, která broadcastuje instrukci ALU jednotkám, každá ALU jednotka buď instrukcí vykoná, nebo nevykoná nic a čeká [24]. V 90. letech byly SIMD systémy rozšířené jako vektorové procesory, nyní to jsou hlavně grafické karty a taky některé části CPU. SIMD je v CPU použito například u Advanced Vector Extensions - AVX instruckí.

SIMD lze efektivně použít například při sčítání dvou polí. Existuje pole x a y o velikosti N, pokud máme N počet ALU jednotek, může se do každé ALU jednotky načíst i-tý prvek pole x[i] a y[i], provést součet a výsledek uložit do pole x[i]. Pokud by systém měl méně ALU jednotek než je velikost pole, rozdělilo by se pole na menší části/dávky. Pro tyto menší dávky by se použil již popsaný postup vícekrát.

Podmínka je, aby každá ALU vykonávala tu samou instrukci nebo nedělala nic. V SIMD musí všechny ALU provádět práci synchronně, vždy musí počkat na broadcast kontrolní jednoty a pak operaci vykonat. ALU si v sobě nemůže uložit instrukci a použit ji později. Architektura je znázorněna na obrázku 6. SIMD se používá v API pro tvorbu počítačové grafiky, například OpenGL, kde se pro reprezentaci objektu interně používají body, přímky a trojúhelníky. OpenGL



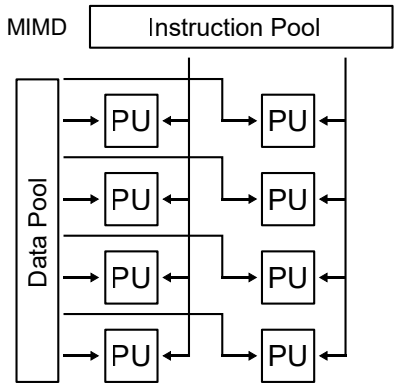
Obrázek 6: Architektura single instruction, multiple data [3]

převádí interní reprezentaci objektů na pole pixelů, které může být zobrazeno na obrazovce počítače. Některé části tohoto procesu jsou programovatelné, tyto části se nazývají shadery. Jsou to většinou krátké funkce v jazyce C a vykonávají se paralelně. Zpracovávají více stejných elementů, jako jsou vertexy, díky tomu může graficka karta pro výpočet využít SIMD architetkuru.

3.1.2 MIMD

MIMD systémy podporují operovat s více různými instrukcemi nad různými daty současné. Tyto systémy jsou většinou sestaveny z procesorových jednotek, jader, které na sobě nejsou závislé a každé z nich má svoji řídicí jednotku a ALU [24]. Na rozdíl od SIMD jsou tyto systémy asynchronní, každý procesor může vykonávat práci svým vlastním tempem, nemusí na nic čekat jako třeba brodcast instrukce u SIMD. Synchronizaci jader musí zařídit sám programátor. I když má každé jádro za úkol vykonávat tu stejnou sekvenci instrukcí, může v jednom čase vykonávat jinou instrukci než ostatní jádra. Schéma architektury je na obrázku 7.

MIMD systémy se rozdělují na dvě části, systémy se sdílenou pamětí a systémy s distribuovanou pamětí. V systému se sdílenou pamětí je každý procesor připojen ke stejné paměti a každý může přistupovat na kteroukoliv pozici v paměti. Pro přístup do paměti se nemusí ostatních procesorů dotazovat. V systému s distribuovanou pamětí má každý procesor svou paměť a tvoří pár, tyto páry spolu komunikují přes síť. Komunikují spolu pomocí zasílaní zpráv nebo pomocí speciálních funkcí, které zajistí přístup do paměti jiného procesoru [25].



Obrázek 7: Architektura multiple instruction, multiple data [3]

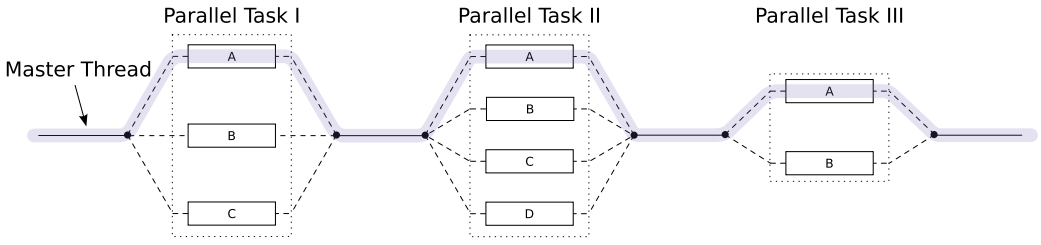
3.2 OpenMP

Open Multi-Processing je API pro paralelní programování se sdílenou pamětí. OpenMP lze použít v programovacích jazycích C, C++ a Fortran, je podporováno na hodně platformách a operačních systémech. Použití je velmi jednoduché, v kódu se používá pomocí directivy #pragma omp společně s nějaký příkazem. Mnohdy ani není potřeba nějak více upravovat existující kód, například při paralelizaci foru viz výpis 1. Stejný kód je použit jak pro paralelní tak i sériovou verzi programu. Programátor tedy pouze specifikuje, co se má udělat, a kompilátor zařídí jak se to provede [26].

```
#pragma omp parallel for
for (int i = 0; i < 100000; i++) {
   a[i] = 2 * i;
}
```

Výpis 1: Přiklad použití OpenMP pro paralelení for

OpenMP je založené na programovacím modelu fork-join. Jedno hlavní vlákno zpracovává všechny sériové sekce. Když hlavní vlákno přijde na paralelní sekci tak vytvoří nové vlákna (fork). Paralelní vlákna pracují na sobě nezávisle a na konci paralelní sekce jsou zase synchronizovány (join), proces je znázorněn na obrázku 8. Každé vlákno pracuje se sdílenými a vlastními proměnnými, které se taky dají definovat pomocí direktiv.



Obrázek 8: Exekuční model fork-join [4]

3.3 CUDA

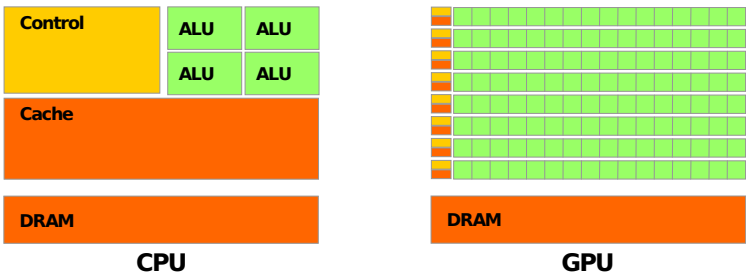
Grafické karty jsou jedním z nejefektivnějších způsobů jak využít paralelismus. Compute Unified Device Architecture (CUDA) je API od nVidie, díky němuž je možné na grafické kartě programovat a počítat obecné věci, přesněji jde o pojem GPGPU (General-Purpose computing on Graphics Processing Units). Konkurencí této technologie jsou například frameworky OpenACC a OpenCL, které taky grafické karty od nVidie podporují. Nejčastěji se s tímto API pracuje v jazyce C/C++, ale existují také wrappery, které umožní použít grafickou kartu třeba v Pythonu, nejčastěji při používání neuronových sítí společně s frameworkem Tensorflow nebo PyTorch.

3.3.1 Programování na GPU

Grafické čipy jsou vyvíjeny, aby byly schopny zpracovat až tisíce vláken zároveň. Všechny tyto vlákna musí být na sobě nezávislá, není totiž zaručeno pořadí, kdy se jednotlivá vlákna vykonají. GPU je založeno na SIMD (ve skutečnosti se jedná o SIMT) architektuře, umí sice zpracovat více vláken najednou, ale každé z nich musí vykonat tu stejnou instrukci [27]. Aby se karta využila co nejefektivněji, při programování je tedy dobré omezit počet podmínek, po kterých může karta v každém vláknu použit jinou instrukci. Karta je optimalizovaná pro sekvenční přístup do paměti.

3.3.2 CPU a GPU porovnání

Na obrázku 9 je znázorněna poměr jednotek v typickém moderním CPU a GPU. Zelené části reprezentují výpočetní jednotky, žluté části jsou kontrolní prvky a oranžové jsou paměti. RAM paměť není přímo součástí GPU nebo CPU, ale je potřebná k ukládání dat. Většina obvodů na grafické kartě určené pro výpočty.



Obrázek 9: CPU vs GPU architektura [5]

Porovnání specifikací grafické karty nVidia Geforce RTX 2080 Ti a procesoru Intel i9-10980XE je v tabulce 1. Z tabulky lze vyčíst, že grafická karta má zhruba 15krát větší teoretický výpočetní výkon než procesor a to při skoro stejné pořizovací ceně.

Tabulka 1: Tabulka porovnání GPU a CPU

	GeForce RTX 2080 Ti	Intel i9-10980XE
Frekvence	1 545 MHz	4 600 MHz
Počet jader (vláken)	4 352	18 (36)
Teoretický výkon	13,45 TFlops	875 GFlops
Velikost paměti	11 GB	max 256 GB
Spotřeba	260 W	165 W
Cena	30 000 Kč	29 000 Kč

3.3.3 CUDA a GPU architektrua

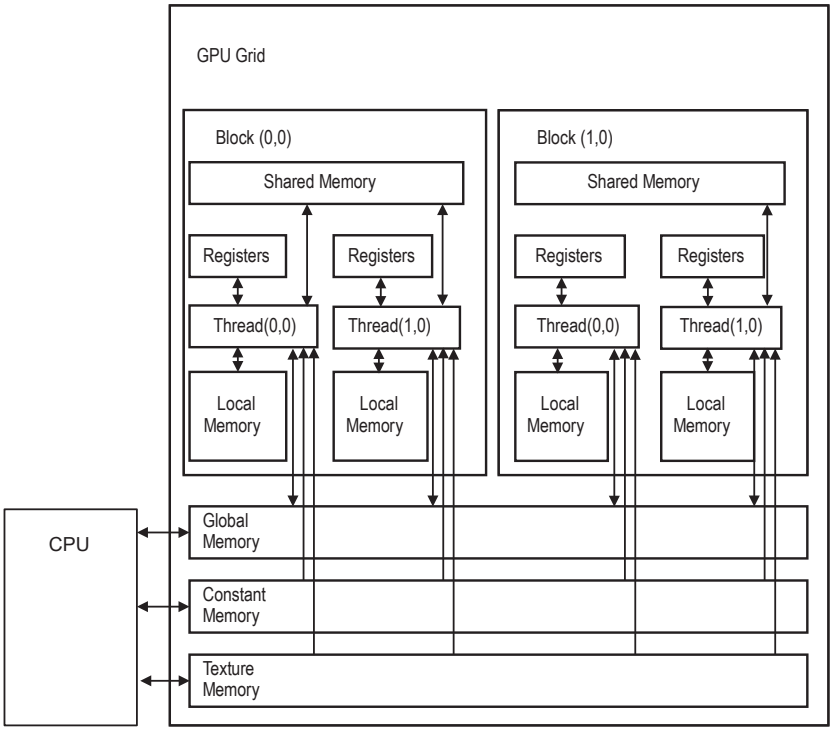
CUDA je postavena na architektuře single instruction multiple threads (SIMT). Karty mají mnoho jader (výpočetní jednotka), kde každé z nich má ALU a FPU pro výpočty, toto jádro se nazývá CUDA core. Jádra jsou seskupeny do skupin zvaných streaming multiprocesory (SM). Multiprocesor vytváří, ovládá, plánuje a spouští vlákna ve skupinách o velikosti 32. Tato skupina vláken se nazývá warp. Kontrolní jednotka SM řídí každé ze svých jader, aby pro každý thread ve warpu vykonavaly tu stejnou instrukci.

V CUDĚ jsou do programu přidány kernely, funkce které poběží na GPU v několika vláknech. Vlákna jsou na kartě seskupena do bloků, kde jeden blok může mít maximálně 1024 vláken. Bloky jsou pak uspořádány do gridu. Počet bloků většinou závisí na konkrétní úloze a na datech, které se zpracovávají [28].

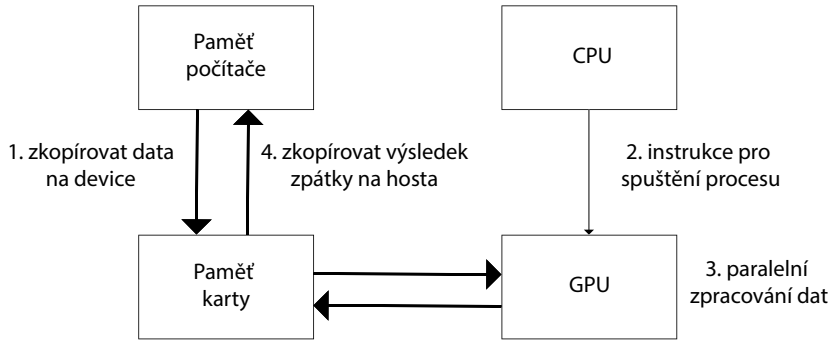
GPU architektura má různé typy pamětí, jako globální (*global*), konstantní (*constant*), texturová (*texture*), sdílená (*shared*), lokální (*local*) a registry. Každé vlákno má pouze svoji lokální paměť a registry, které nesdílí s jinými vlákny. Na úrovni bloků je sdílená paměť, do které mají přístup všechny vlákna daného bloku, přes tuto paměť si můžou vlákna předávat data. Na úrovni karty je texturová a konstantní paměť, která se používá pro read-only data. Globální paměť je přístupná pro GPU tak i CPU, slouží pro přenos dat mezi počítačem a kartou [29]. Rozložení pamětí je možno vidět na obrázku 10.

3.3.4 Proces CUDA výpočtu

Nejprve se na kartě musí alokovat místo pro vstupní data, většinou se jedná o pole. Pokud je potřeba, tak se alokuje prostor i pro výstupní data, někdy je možné výstup uložit přímo do vstupního pole. Když je paměť alokovaná, je možné přenést data z hosta (CPU a jeho pamět) na device (GPU a jeho paměť). Jak jsou data převedena na kartu, host zavolá kernel, který na kartě spustí vlákna. Vlákna vykonají svůj úkol a uloží svůj výsledek do předem alokovaného pole na kartě. Výsledek se nakonec zkopíruje zpátky na hosta, kde se dál zpracuje. Proces je znázorněn v diagramu 11.



Obrázek 10: Hierarchie pamětí na GPU [6]



Obrázek 11: Digram procesu CUDA výpočtu

3.3.5 Orientace v gridu

V každém kernelu jsou k dispozici vestavěné proměnné threadIdx (pozice threadu v bloku), blockIdx (pozice bloku v gridu), blockDim (dimenze bloku), gridDim (dimenze gridu). Podle těchto proměnných je možné zjistit ve kterém vlákně nebo bloku se zrovna program nachází, a podle těchto hodnost se pohybovat ve vstupním a výstupním poli. Každá tato proměnná je struktura typu dim3 nebo uint3, takže hodnoty, které se používají k výpočtu, jsou až v členech xyz [30].

Jednoduchý příklad jak se gridu orientovat je ve výpise 2. Kde na řádku 1 je z vestavěných proměnných zjištěna pozice v poli. Pokud je velikost vstupního pole vetší než počet vláken v gridu, tak proměnná idx nepokryje celé pole a je potřeba se posunout o celkový počet vláken v gridu, viz řádek 2 a 7.

```
1 uint32_t idx = blockDim.x * blockIdx.x + threadIdx.x;
2 uint32_t offset = gridDim.x * blockDim.x;
3
4 while (idx < population_size) {
5    ouptup[idx] = input[idx]*258;
6
7    idx += offset;
8 }
```

Výpis 2: Příklad orientace v gridu

4 Návrh a implementace vybraných bio-inspirovaných výpočtů

V této práci byly implementovány tři evoluční algoritmy. V následující kapitole budou popsány algoritmy optimalizace pomocí roje částic (PSO), diferenciální evoluce (DE) a optimalizace mravenčí kolonii (ACO). U PSO bude popsán rozdíl mezi sériovou a paralelní implementací tohoto algoritmu, a kde mohou nastat problémy. Dále bude popsána funkce na hledání nejlepšího jedince v populaci, paralelní sken, která se používá u všech těchto algoritmů. Nakonec bude popsána vizualizace částic PSO, důvod proč se algoritmus musel pro vizualizaci uměle zpomalit a popis stolu, kde bude vizualizace použita.

4.1 Implementace optimalizace pomocí roje částic

V této práci byla implementovaná verze PSO, kde je použitý elitismus, velikost skupiny (sousedů) se bere jako všechny částice, je tedy pouze jedna skupina, a rychlost je v každém kroku snižována setrvačností. Pohyb každé částice je ovlivněn svou vlastní nejlepší známou pozicí, a také nejlepší známou pozicí z celé populace za její historii (elitismus). Nejlepší známé pozice se aktualizují při tom, jak se částice pohybují a hledají nové řešení. Tyto pravidla by měly pohybovat populací k nejlepšímu řešení.

4.1.1 Algoritmus implementovaného PSO

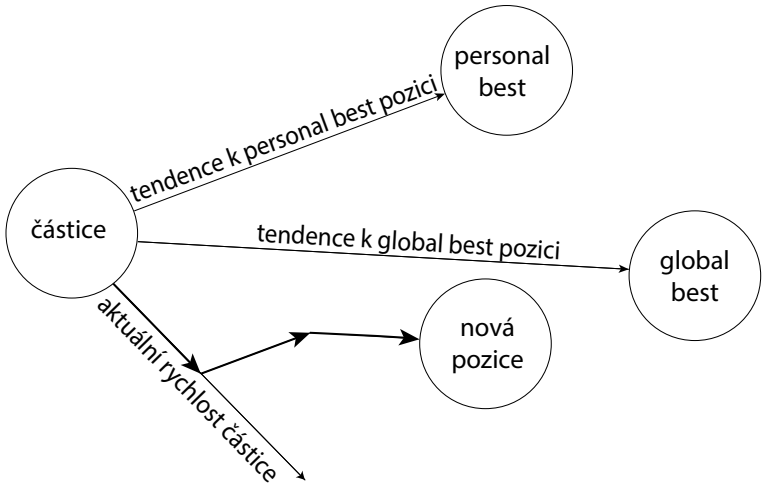
Na začátku se vygeneruje nová populace uniformě rozprostřená po hledacím prostoru. Každá částice se ohodnotí podle účelové funkce a ta stejná hodnota je nastavená jako pBest. Po té se v celé populaci najde to nejlepší řešení a nastaví se jako gBest. Pro každou částici se dále nastaví počáteční rychlost, směr, kterým se bude částice dále pohybovat. Maximální rychlost by neměla být vyšší než vzdálenost mezi hranicemi vyhledávacího prostoru.

Při pohybu částic se pro každou dimenzi v každé částici vygenerují dvě náhodné čísla a Většina programů, které programátoři píší, jsou sériové programyvypočítá se nová rychlost podle vzorce 1 a následně nová pozice podle vzorce 2 [31]. Pohyb částice je znázorněn na obrázku 12. Hned po získání nové pozice se částice ohodnotí účelovou funkcí a zkontroluje se, zda má nová pozice lepší výsledek než její pBest, při lepší hodnotě se zapíše nová pozice jako pBest. To samé porovnání se udělá i s gBest hodnotou.

$$v_d(t+1) = w \cdot v_d(t) + c_1 r_1 (pBest_d - x_d(t)) + c_2 r_2 (gBest_d - x_d(t)) \tag{1}$$

$$x_d(t+1) = v_d(t) + x_d(t+1) \tag{2}$$

kde:



Obrázek 12: Pohyb částice ovlivněný pBest a gBest

$v_d(t+1)$ = rychlost částice v dalším kroku
$v_d(t)$ = rychlost částice v tomto kroku
$x_d(t+1)$ = pozice částice v dalším kroku
$x_d(t)$ = pozice částice v tomto kroku
$pBest_d$ = nejlepší známá pozice dané částice
$gBest_d$ = nejlepší známá pozice celé populace
c_1, c_2 = učící parametry
r_1, r_2 = náhodná čísla mezi 0 a 1
w = setrvačnost

Tento cyklus se opakuje víckrát, dokud není splněna podmínka pro konec, což je většinou počet iterací. V každé iteraci se mění parametr setrvačnosti w pode vzorce 3. Algoritmus je také popsán pseudokódem 2.

$$w = w_{start} - \frac{(w_{start} - w_{end}) \cdot m}{M} \tag{3}$$

kde:

w_{start} = setvačnost na začátku
w_{end} = setrvačnost na konci
m = číslo migrace
M = celkový počet migrací

Algorithm 2 Pseudokod pro sériový PSO.

1: **Vstup:**
   velikost populace N
   dimenze problému D
   počet migrací M
   tendence konvergence k pBest c_1 a gBest c_2
   maximální rychlost v_{max}
   rychlost první iterace v_{start}
   rychlost poslední iterace v_{end}
   fitness funkce $f(x)$
2: **Inicializace:**
   náhodná populace jedinců $x_i, i \in$[1,N]
   vector rychlosti jedinců $v_i, i \in$[1,N]
   nastavit personal best pozici jedince $pBest_i \leftarrow x_i, i \in$[1,N]
   nastavit nejlepší pozici všech jedinců $gBest \leftarrow argmin(pBest_i), i \in$[1,N]
3: **for** $m = 1$ to M **do**
4: $w = w_{start} - ((w_{start} - w_{end}) \cdot m)/M$
5: **for** $i = 1$ to N **do**
6: **for** $j = 1$ to D **do**
7: vygenerovat náhodný vector r_1 a r_2
8: $v_{i,j} = w \cdot v_{i,j} + c_1 r_1 (pBest_{i,j} - x_{i,j}) + c_2 r_2 (gBest_j - x_{i,j})$
9: $x_{i,j} = v_{i,j} + x_{i,j}$
10: **end for**
11: **if** $f(pBest_i) > f(x_i)$ **then**
12: $pBest_i \leftarrow x_i$
13: **if** $f(gBest) > f(x_i)$ **then**
14: $gBest \leftarrow x_i$
15: **end if**
16: **end if**
17: **end for**
18: **end for**

4.1.2 Paralelní verze

Při paralelizaci tohoto algoritmu vzniká při kontrole a aktualizaci nejlepší pozice gBest kritická sekce. Zatímco jedno vlákno pozici gBest čte, jiné ho může upravovat. Je nutné aktualizaci gBest posunout na později, až kdy jsou všechny částice posunuté na nové místo a je aktualizován jejich pBest. Až jsou tyto kroky provedeny, najde se ve všech pBest nová pozice gBest. Toto posunutí má ale za následek, že stejný počet iterací dojde k horšímu výsledku než sériová verze. Pokud seriová verze najde nový gBest hned u první částice, můžou další částice v té stejné iteraci počítat již novým a lepším gBestem, částice tak rychleji konvergují k lepšímu výsledku. Sériová verze sice má lepší výsledek při stejném počtu iterací, ale paralelní verze je mnohonásobně rychlejší, aby se ztráta dohnala, stačí spočítat jednu iteraci navíc, výsledek bude lepší a časově je na tom paralelní verze stále lépe, i když vypočítala o jednu iteraci více. Paralelní algoritmus lze najít v pseudokódu 3.

Algorithm 3 Pseudokod pro paralelní PSO.

1: **Vstup:**
velikost populace N
dimenze problému D
počet migrací M
maximální rychlost v_{max}
tendence konvergence k pBest c_1 a gBest c_2
rychlost první iterace v_{start}
rychlost poslední iterace v_{end}
fitness funkce $f(x)$
2: **Inicializace:**
náhodná populace jedinců $x_i, i \in$[1,N]
vector rychlosti jedinců $v_i, i \in$[1,N]
nastavit personal best pozici jedince $pBest_i \leftarrow x_i, i \in$[1,N]
nastavit nejlepší pozici všech jedinců $gBest \leftarrow argmin(pBest_i), i \in$[1,N]
3: **for** $m = 1$ to M **do**
4: $w = w_{start} - ((w_{start} - w_{end}) \cdot m)/M$
5: **for** $i = 1$ to N **do**
6: **for** $j = 1$ to D **do**
7: vygenerovat náhodné čísla r_1 a r_2
8: $v_{i,j} = w \cdot v_{i,j} + c_1 r_1 (pBest_{i,j} - x_{i,j}) + c_2 r_2 (gBest_j - x_{i,j})$
9: $x_{i,j} = v_{i,j} + x_{i,j}$
10: **end for**
11: **end for**
12: **for** $i = 1$ to N **do**
13: **if** $f(pBest_i) > f(x_i)$ **then**
14: $pBest_i \leftarrow x_i$
15: **end if**
16: **end for**
17: $gBest \leftarrow argmin(pBest_i), i \in$[1,N] (paralelní redukce)
18: **end for**

4.2 Implementace diferenciální evoluce

Diferenciální evoluce (DE) je technika, která se iterativně snaží optimalizovat své řešení. DE optimalizuje problém svou populací kandidátních řešení, které se mezi sebou kombinují na základě jednoduchých pravidel a vytváří tak nové kandidátní řešení, nové jedince. Nový jedinec se ohodnotí podle účelové funkce a na základě této hodonty se rozhodne, zda se zachová původní jedinec nebo se nahradí novým. Existuje mnoho různých variant DE, tato práce používá variantu *DE/rand/1/bin*. V algoritmech při paralelizaci nikde nevzniká žádná kritická sekce, lze tedy jednoduše paralelizovat.

4.2.1 Algoritmus DE/rand/1/bin

Vygeneruje se uniformě rozdělená populace v rozsahu definičního oboru účelové funkce. Velikost populace závisí na vstupních parametrech. Pro každého jedince se vyberou z populace další 3 jedinci. Znamená to, že k vytvoření jednoho potomka je potřeba 4 rodičů. Z těchto 3 náhodně vybraných jedinců se vypočítá podle vzorce 4 šumový vektor v. Tento proces se nazývá mutace.

$$v = x_{r3} - F \cdot (x_{r1} - x_{r2}) \tag{4}$$

kde:

$x_{r1,r2,r3}$ = 3 náhodně vybraní jedinci z populace
v = šumový vektor
F = mutační konstanta

Dále se provede křížení. Na rozdíl od obecného evolučního algoritmu v DE křížení probíhá až po mutaci. Křížení je zkombinování původního jedince a šumového vektoru, ze kterých vznikne trial vektor. Pro každou dimenzi problému se vygeneruje náhodné číslo, pokud je toto číslo menší než parametr CR, do trial vektoru se použije hodnota z šumového vektoru, pokud ne tak se použije původního jedince. Trial vektor se ohodnotí účelovou funkcí a jeho hodnota se porovná s hodnotou původního jedince. Pokud je hodnota trial vektoru lepší než původní jedinec, tak jej trial vektor nahradí. Celý tento proces se několikrát opakuje. Na konci všech iterací se z poslední generace vybere to nejlepší řešení. Algoritmus je popsán v pseudokódu 4.

Při paralelní verzi jede každý výpočet pro jednoho jedince ve svém vlastním vlákně. V paralelní verzi na grafice může docházet ke zpomalení pří náhodném výběru 3 různých vektorů, protože unikátního výběr je zde ošetřen pomocí podmínek.

4.3 Implementace optimalizace mravenčí kolonii

V této práci je implementována základní verze algoritmu Ant Colony, kterou autoři nazývali *ant-cycle* [32], kde se feromony aktualizují až poté, co všichni mravenci dokončí svou cestu. V jejich knize také mají verze jménem *ant-density* a *ant-quantity*, které aktualizují feromony při každém kroku, tyto verze ale měly horší výsledky než *ant-cycle*. Implementace je otestována na problému obchodního cestujícího.

4.3.1 Problém obchodního cestujícího

Problém obchodního cestujícího má za úkol najít nejkratší cestu v množině měst. Obchodní cestující začne svou cestu v kterémkoliv z těchto měst, všechny ostatní města projde pouze jednou a svou cestu ukončí v městě, ve kterém původně začal. Formálně zadáno: najít nejkratší Hamiltonovskou cestu v úplném grafu. Problém patří do skupiny NP-úplných problémů, neexistuje pro něj žádný polynomiální deterministický algoritmus.

Algorithm 4 Pseudokod pro DE/rand/1/bin

Vstup:
velikost populace N
dimenze problému D
počet generací G
práh křížení CR
mutační konstanta F
fitness funkce $f(x)$
Inicializace:
náhodná populace jedinců $x_i, i \in$[1,N]
pole pro novou populaci $newPop_i, i \in$[1,N]
for $g = 1$ to G **do**
  for $i = 1$ to N **do**
    náhodně vybrat 3 různé jedince v populaci x_{r1}, x_{r2}, x_{r3}
    $randJ \leftarrow$ náhodné celé číslo $\in [1, N]$
    $v = x_{r3} + F * (x_{r1} - x_{r2})$
    for $j = 1$ to D **do**
      if rand() < CR || j==randJ **then**
        $trialVektor_j = v_j$
      else
        $trialVektor_j = x_j$
      end if
    end for
    if $f(trialVektor) < f(x_i)$ **then**
      $newPop_i = trialVektor$
    else
      $newPop_i = x_i$
    end if
  end for
  vyměnit starou populaci za novou $swap(x, newPop)$
end for
$gBest \leftarrow argmin(x_i), i \in$[1,N] (paralelní redukce)
return $gBest$

4.3.2 Algoritmus implementovaného ACO

Na začátku algoritmu se ke každému městu (uzlu v grafu) přiřadí jeden mravenec, počet mravenců a měst je stejný. Ze všech měst, která zatím nebyly mravencem navštíveny, si další město vybírá podle vzorce 5. Ten každému městu přiřadí váhu, která určuje pravděpodobnost výběru.

$$p_{i,j} = \frac{[\tau_{i,j}]^\alpha [\eta_{i,j}]^\beta}{\sum_{l \in N} [\tau_{i,l}]^\alpha [\eta_{i,l}]^\beta} \tag{5}$$

kde:

$\tau_{i,j}$ = tabulka feromonů mezi uzly grafu
$\eta_{i,j}$ = převrácená vzdáleností mezi uzly grafu $\eta_{i,j} = 1/vzdálenost_{i,j}$
α = vliv feromonů na pravděpodobnost pro výběr dalšího uzlu
β = vliv vzdálenosti na pravděpodobnost pro výběr dalšího uzlu
N = města, které mravenec zatím nenavšívil

Poté co všichni mravenci dokončili svou cestu, se aktualizuje tabulka feromonů podle vzorce 6. První se všechny feromony vynásobí hodnotou $(1 - \rho)$, což je evaporace, a poté každý mravenec přidá na cestu, kterou prošel svoje feromony. Množství feromonů, který každý mravenec na svou cestu aplikuje, závisí na vzdálenosti jeho ušlé cesty, čím kratší jeho cesta byla, tím víc feromonů bude přidáno.

$$\tau_{i,j} = (1 - \rho)\tau_{i,j} + \sum_{k=1}^{m} \Delta\tau_{i,j}^{k} \tag{6}$$

kde:

ρ = rychlost evaporace feromonů
m = je počet mravenců

$$\Delta\tau_{i,j}^{k} = \begin{cases} 1/L^{k} & \text{pokud hrana grafu (cesta) byla použita mravencem k} \\ 0 & \text{jinak} \end{cases} \tag{7}$$

kde:

L = vzdálenost, kterou mravenec ušel

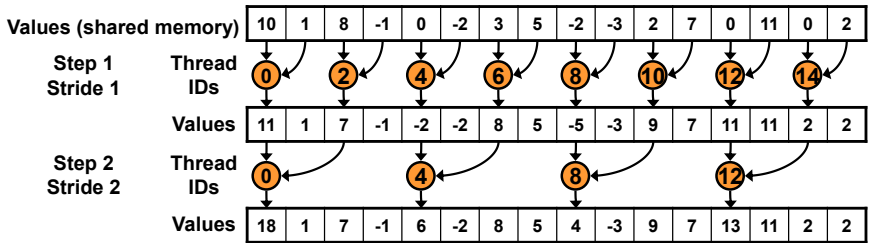
Hrany, které byly použity více mravenci, budou mít větší hodnotu, takže bude větší pravděpodobnost, že si tuto cestu mravenec příště vybere.

4.4 Paralelni sken, hledání minima

Hledání nejlepších jedinců v populaci se v každém algoritmu provádělo pomocí paralelního skenu. Při paralelním skenu se načtou data z globální paměti do sdílené paměti. Přístup k sdílené paměti je rychlejší, než pro každý prvek přistupovat do globální paměti. Grafická karta je optimalizovaná pro sekvenční přístup do paměti, čte vždy více prvků vedle sebe, a může si tak vedlejší prvky uložit od paměti cache, při dotazu na vedlejší prvek se už bude stačit doptat na hodnotu do pamětí cache, což je rychlejší než jít zase do sdílené paměti. Bylo potřeba optimalizovat přístup do sdílené paměti, aby prvky, co se načítají v jednom kroku, byly co nejblíže u sebe. Toto bylo zajištěno změnou stride indexací za threadId idexaci v cyklu, který prochází prvky v sdílené paměti. Příklad kódu stride indexace je ve výpise 3 a indexace pomoci threadId je ve výpise 4. Znázornění rozdílu jejich přístupu do paměti je vidět na obrázcích 13 a 14.

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {
   int index = 2 * s * tid;
   if (index < blockDim.x) {
      if (sdata[index] > sdata[index + s]) {
         sdata[index] = sdata[index + s];
      }
   }
   __syncthreads();
```

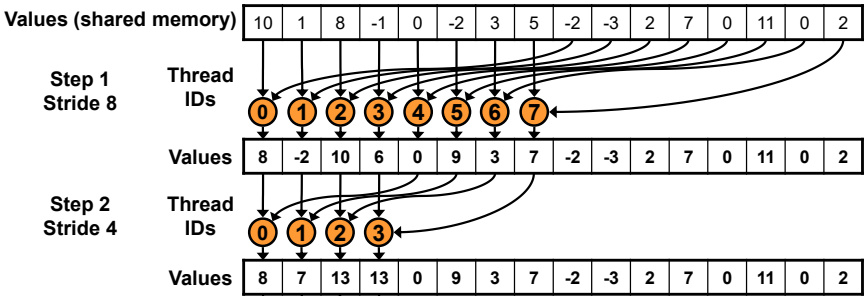
Výpis 3: Přiklad stride indexace



Obrázek 13: Nesekvenční (*interleaved*) přístup k paměti [7]

```
for (unsigned int next = blockDim.x >> 1; next >= 1; next >>= 1) {
   if (tid < next) {
      if (sdata[tid] > sdata[tid + next]) {
         sdata[tid] = sdata[tid + next];
      }
   }
   __syncthreads();
}
```

Výpis 4: Přiklad threadId indexace



Obrázek 14: Sekvenční přístup k paměti [7].

Další optimalizace paralelního skenu byla rozvinutí foru pomocí makra #pragma unroll, tímto je možné se vyhnout častého vyhodnocování podmínky v cyklu for, a ušetří se trochu času. Dále je zbytečené všechny prvky přeukládat do shared paměti. Při prvním načtení můžeme oba prvky načtené z global paměti rovnou porovnat a uložit do shared paměti pouze výsledek porovnání.

Další problém v hledání částice s minimální hodnotou je, že se nejedná úplně o čistý paralelní sken. Podmínka se sice vyhodnocuje na základě jedné hodnoty, ale tato hodnota patří k nějaké částici, která má taky souřadnici. Bylo potřeba si po porovnání uložit i pozici částice a ne jenom minimální hodnotu. V prvních verzích mé implementace byly přeukládany přímo hodnoty pozice částice. S tímto řešením se brzo objevil problém s nedostatkem sdílené paměti. Kvůli tomu se implementace změnila a přeukládá se pouze ukazatel na místo, kde se pozice částice nachází, a ne samotné hodnoty. Ve verzi, kde se přeukládaly všechny hodnoty pozice, byl požadavek na velikost sdílené paměti vyšší, než karta GTX 1070 zvládne, paměť nestačila již při problémech s dimenzi 47.

Původně bylo vše vyvíjeno jako template, aby evoluční algorimty mohly pracovat jak s datovým typem float tak i double. Při implementaci paralelního skanu se ale přišlo na to, že sdílená paměť nepodporuje datový typ double, takže EA jsou implementované jako templaty, ale ve skutečnosti je možné použit pouze datový typ float.

```
1  template< class T>
2  __global__ void find_gbest(
3     T* __restrict__ dPersonalBestPositions,
4     T* __restrict__ dPersonalBestValues,
5     T* __restrict__ global_best,
6     T* __restrict__ global_best_value,
7     const uint32_t population_size,
8     const uint32_t dimension)
9  {
10    extern __shared__ T sharedMemory[];
11    T* sharedValues = sharedMemory;
12    T** sharedPositions = (T**)&sharedValues[blockDim.x];
13
14    uint32_t idx = blockDim.x * blockIdx.x + threadIdx.x;
15    uint32_t tid = threadIdx.x;
16    uint32_t offset = gridDim.x * blockDim.x;
17    uint32_t offsetDim = offset * dimension;
18    uint32_t offset2x = offset << 1;
19
20    T* val_in_shared1;
21    T* val_in_shared2;
```

```
22    T** pos_in_shared;
23    T first_val;
24    T second_val;
25    uint32_t idx_first_val = 0;
26    uint32_t idx_second_val = 0;
27    uint32_t idx_first_pos = 0;
28    uint32_t idx_second_pos = 0;
29
30
31    while (idx < population_size) {
32       val_in_shared1 = &sharedValues[tid];
33       pos_in_shared = &sharedPositions[tid];
34
35       idx_first_val = idx;
36       idx_first_pos = idx * dimension;
37
38       first_val = dPersonalBestValues[idx_first_val];
39       second_val = dPersonalBestValues[idx_first_val + blockDim.x];
40
41       if (first_val < second_val) {
42          *val_in_shared1 = first_val;
43          *pos_in_shared = &dPersonalBestPositions[idx_first_pos];
44       }
45       else {
46          *val_in_shared1 = second_val;
47          *pos_in_shared = &dPersonalBestPositions[idx_first_pos + offsetDim];
48       }
49       __syncthreads();
50
51       #pragma unroll 4
52       for (unsigned int next = blockDim.x >> 1; next >= 1; next >>= 1) {
53          if (tid < next) {
54             val_in_shared2 = val_in_shared1 + next;
55
56             if (*val_in_shared1 > * val_in_shared2) {
57                *val_in_shared1 = *val_in_shared2;
58                *pos_in_shared = *(pos_in_shared + next);
59             }
60          }
```

```
61          __syncthreads();
62       }
63
64       if (tid == 0) {
65          if (*global_best_value > * val_in_shared1) {
66             *global_best_value = *val_in_shared1;
67             copy_values_in_dimension<T>(global_best, *pos_in_shared, dimension);
68          }
69       }
70       __syncthreads();
71
72       idx += offset2x;
73    }
74 }
```

Výpis 5: Implementace paralelního skanu pro hledání nejlepšího jedince

Ve výpise 5 je implementace hledání nejlepšího jedince pomocí paralelního skanu. Na řádcích 10 až 12 je rozdělení sdílené paměti na dvě části, do jedné se budou ukládat hodnoty jedinců a do druhé části pouze ukazatele na pozice jedinců. Na řádcích 14 až 18 jsou proměnné pro pohybování se v poli, které se vypočítají na základě vestavěných proměnných. Dále se na řádcích 38 a 39 načtou hodnoty jedinců z globální paměti, na řádku 41 se vyhodnotí, která z nich je menší a podle toho se lepší hodnota uloží do první části sdílené paměti, a do druhé části paměti se uloží pouze odkaz na globální pamět. K hodnotám tohoto odkazu se bude přistupovat až na konci, takže opakovaný přístup do globální paměti nevznikne a nezpomalí program. Na řádku 51 je direktiva na rozložení foru, jedna z optimalizací, dále následuje for, který zajistí sekvenční přístup do sdílené paměti. V tomto foru se již pracuje pouze se sdílenou pamětí. Tento for zajistí, že v na první pozici ve sdílené paměti, bude hodnota nejlepšího prvku a odkaz na pozici nejlepšího jedince. Na řádcích 64 až 65 se nakonec pouze v jednom vlákně porovná hodnota s aktuální hodnotou nejlepšího prvku, jehož výchozí hodnota je nastavena na největší možnou hodnotu datového typu float. Pokud je tato podmínka splněna, přepíše se hodnota nejlepšího prvku na novou hodnotu ze sdílené paměti a také se překopírujou hodnoty pozice jedince z globální paměti do pozice nejlepšího jedince. Nakonec se na řádku 72 posune index a začne se hledat nová nejlepší hodnota v další části pole.

4.5 Visualizace částic

Jedním z úkolů bylo pohyb částic u PSO vizualizovat. Vizualizovalo se pouze PSO, protože u tohoto EA jsou nějaké částice, které se hýbou, simulují pohyb. Například u diferenciální evoluce žádný pohyb není a tak není co vizualizovat. Pro znázornění částic bylo použito OpenGL. Při první implementaci se částice pohybovaly po ploše moc rychle, a nebylo možné poznat, kam

se která částice pohybuje, bylo potřeba pohyb uměle zpomalit. Omezení maximální rychlosti PSO nebylo možné, protože při moc malé rychlosti algoritmus nebyl schopen korektně fungovat. Použila se tedy normální rychlost, ale pohyb z jednoho bodu do druhého byl zpomalen interpolací mezi migracemi.

4.5.1 Propojení OpenGL s CUDA

Informace o částicích již jsou na grafické kartě, není je tedy za potřebí znova posílat z hosta. Aby OpenGL bylo schopné data z CUDY vyčíst, je potřeba nejprve VAO buffery registrovat pomocí funkce cudaGraphicsGLRegisterBuffer a poté je namapovat na stejné pole, kde jsou data o částicích uloženy pomocí cudaGraphicsResourceGetMappedPointer.

Pokud jsou pole s daty správně namapována, je možné je použít ve vertex shaderu, kam přijde pozice částice jako datový typ vec2 a její hodnota jako float. Souřadnicím XY se musí upravit měřítko, protože souřadnice v OpenGL jsou jiné, než doména na které částice hledaly minimum. Souřadnice Z je kromě pozice dále použita pro barva částice, aby byl vidět rozdíl ve funkční hodnotě částice.

Částice jsou vykreslovaný pomoci spritů, je to jeden z nerychlejších způsobů jak vykreslit velké množství kuliček. Jako parametr je zde zadán poloměr kuličky. Testuje se, jestli daný pixel pixel náleží do kruhu podle vzdálenosti od středu. Pokud je pixel mimo kuličku, nic se s pixelem ve fragment shaderu neděje. Pokud je uvnitř kruhu, spočítá se v bodě ještě labert shader, aby tento sprite vypadal více prostorově. Zobrazené částice je možné vidět na obrázku 18 jak hledají minimum na sphere funkci.

4.5.2 Interpolace

Jelikož při normální rychlosti částice u PSO na obrazovce jenom skákaly a nebylo možné z nich nic vyčíst a omezením maximálni rychlosti se stal algoritmus nefunkční, muselo se umělé zpomalení udělat jinak. Nejvíc vyhovující řešení byla interpolace mezi starou a novou pozicí částice mezi migracemi.

```
float amount = (float)stepsCount / (float)maximumSteps;

dInterpolatedPosition[idx] =
   (1.0f - amount) * dPreviousPositions[idx] + (amount)*dCurrentPositions[idx];
```

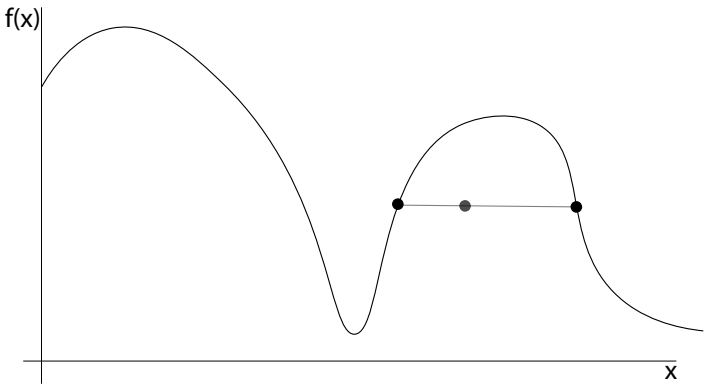
Výpis 6: Výpočet pro interpolaci pozice

Jelikož jsou všechny pozice částic uloženy na kartě, tak i interpolace musí probíhat na kartě. PSO si v sobě neuchovává pozice minulé migrace, takže než se spustí nová migrace PSO, musí si třída s Interpolací udělat kopii pole s aktuálními pozicemi všech částic, až poté spustit migraci částic na nové místo. V interpolaci se nastaví množství interpolačních kroků, a reálná migrace

se spouští až po tom co se tyto kroky provedou. Nová mezipozice se vypočítá pomocí kódu ve výpise 6.

Po první verzi implementace nastal problém. Pokud nějaká částice na povrchu přeskočila vyvýšeninu, znázorněno na obrázku 15, zobrazovala se nová interpolovaná částice pod povrchem a nebylo ji možné vidět. Hodnotu z částice interpolovat není možné, takže bylo nutné získat novou hodnotu pro interpolovanou částici z hloubkové mapy.

Tato nová interpolována pozice je pouze pro vizualizaci. Může se stát, že interpolovaná pozice může projít přes globální minimum funkce, nicméně ve skutečnosti toto místo částice z PSO zatím nenavštívila, protože ho například přeskočila. I když je tato hodnota lepší než dosud nejlepší nalezená hodnota, nemá na výpočet PSO žádný vliv.



Obrázek 15: Interpolovaná částice pod povrchem.

4.5.3 Vizualizace na stole s umělým pískem

Jedno z míst, kde bude vizualizace využita je stůl s umělým pískem, hloubkovou kamerou, která povrch písku snímá a projektorem, který vykreslí částice na umělý písek. Ke snímání povrchu písku se používá hloubková kamera Kinect v2. Hloubkové data z této kamery se pošlou do počítače jako pole, toto pole je nahráno na kartu. Data na krajích tohoto pole se musí oříznout, protože nezabírají přesně pouze povrch stolu, ale i povrch mimo stůl, globální minimum by tak bylo vždy mimo stůl někde na zemi. Tyto oříznuté data se využívají jako účelová funkce pro PSO. Získání hodnoty účelové funkce je zde velmi rychlé, protože v účelové funkci neprobíhá žádný složitý výpočet, pouze se přečte hodnota z pole. Písek na stole vytváří umělou krajinu, na které se hledá nejnižší bod. Na tento stůl již existují jiné aplikace, jako je například simulace vody. Stůl s pískem, kamerou a projektorem je na obrázku 16.

4.5.4 Hloubková kamera Kinect v2

Kinect v2 používá pro měření vzdálenosti dobu letu paprsku světla (*time of flight*). Má v sobě infračervený projektor, který vysílá modulované infračervené světlo, což je poté zachyceno dalším

Obrázek 16: Stůl s hloubkovou kamerou kinect a umělým pískem

senzorem. Odražené infračervené světlo od bližších předmětů bude mít menší dobu letu, než předměty co jsou od senzoru dál. Senzor odražené světlo nasnímá a vyhodnotí, jak moc se modulovaný signál změnil. Měření pomocí doby letu je přesnější a rychlejší na výpočet, než vyhodnocování na bázi strukturovaného světla (*structured light*), kterou používala stará verze Kinectu. Takže může zpracovat více snímků za sekundu.

Kinect v2 má rozlišení hloubkové kamery 512×424 pixelů a zorné pole $70.6° \times 60°$, což vychází zhruba 7×7 pixelů na jednotku úhlu. Každý pixel obsahuje hodnotu vzdálenosti od kamery. Předměty, které jsou před senzorem až moc blízko, můžou být pro senzor až moc jasné a nemusí být vůbec zachyceny [33].

5 Experimenty

Obsahem této kapitoly jsou výsledky testů implementovaných algoritmů na známých problémech. Pro otestování výkonosti PSO a DE se použily známé testovací funkce pro optimalizaci/-matematické vzorce. Test ACO byl proveden na známých datasetech pro problém obchodního cestujícího. Jsou zde porovnány časy běhů algoritmů mezi implementacemi v Pythonu, C++, OpenMP a CUDĚ, a také jak moc závisí velikost problému na zrychlení. Nakonec je otestována aplikace s vizualizací, jak moc částic je schopná v jakém čase zobrazovat. Všechny tyto testy byly provedeny na procesoru Intel Core i7-7700K a grafické kartě nVidia GeForce GTX 1070.

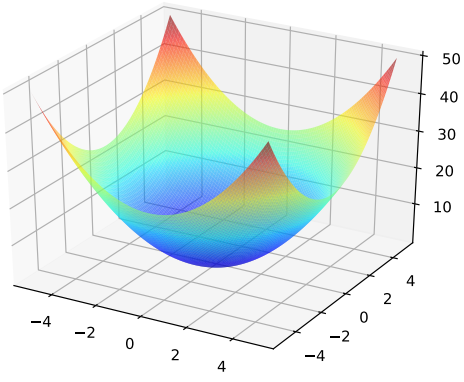
5.1 Vybrané funkce pro testování

Testovací funkce pro optimalizaci si lze představit jako umělé povrchy, lze na nich testovat konvergenci jedinců, přesnost, robustnost algoritmů a jejich výkonost. Pro testování funkčnosti v této práci bylo vybráno následujících 6 funkcí. U všech těchto funkcí je předem známo globální optimum, se kterým se porovná výsledek z evolučních algoritmů a vyhodnotí se správnost nalezeného řešení. Jsou zde zastoupený jak unimodální a tak multimodální testovací funkce. U multimodálních funkcí můžou jedinci uvíznout v lokálních extrémech, a nenaleznou tak nejlepší výsledek. Tento problém je možné vyřešit nastavením parametrů algoritmů. Grafy funkcí jsou zde zobrazeny pouze ve 2 dimenzích, avšak pro evoluční algoritmy je možné funkce použít i ve vyšších dimenzích.

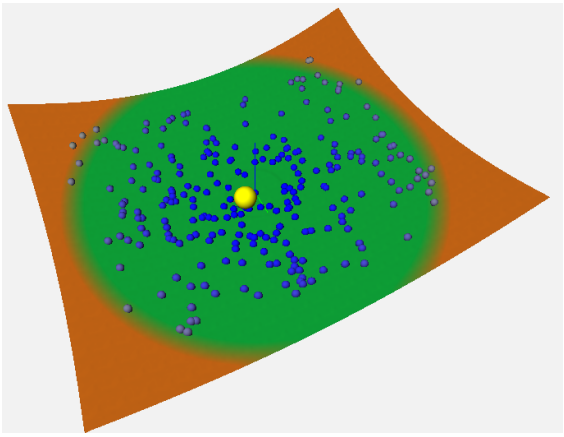
5.1.1 Sphere funkce

$$f(\boldsymbol{x}) = \sum_{i=1}^{n} x_i^2 \tag{8}$$

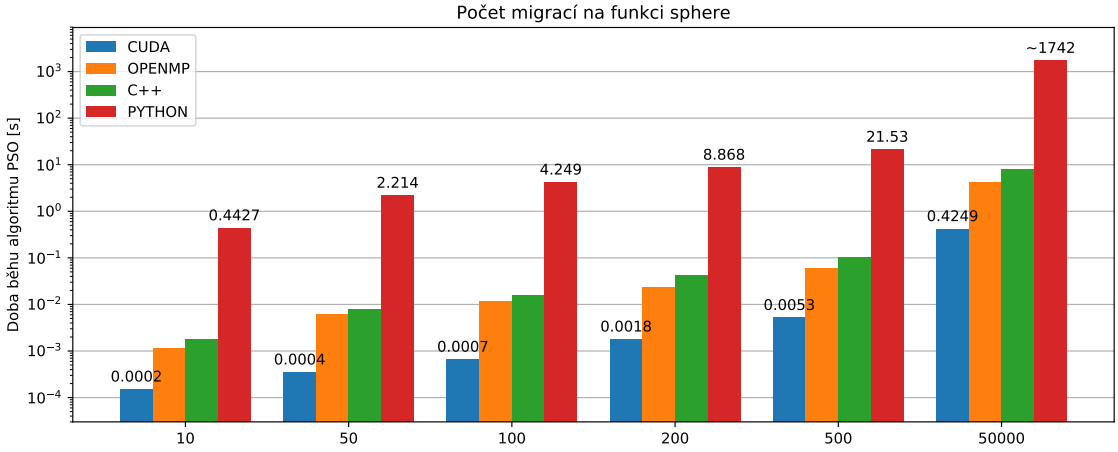
Globální minimum: $f(0, \ldots, 0) = 0$
Definiční obor pro hledání: $-\infty \leq x_i \leq \infty$



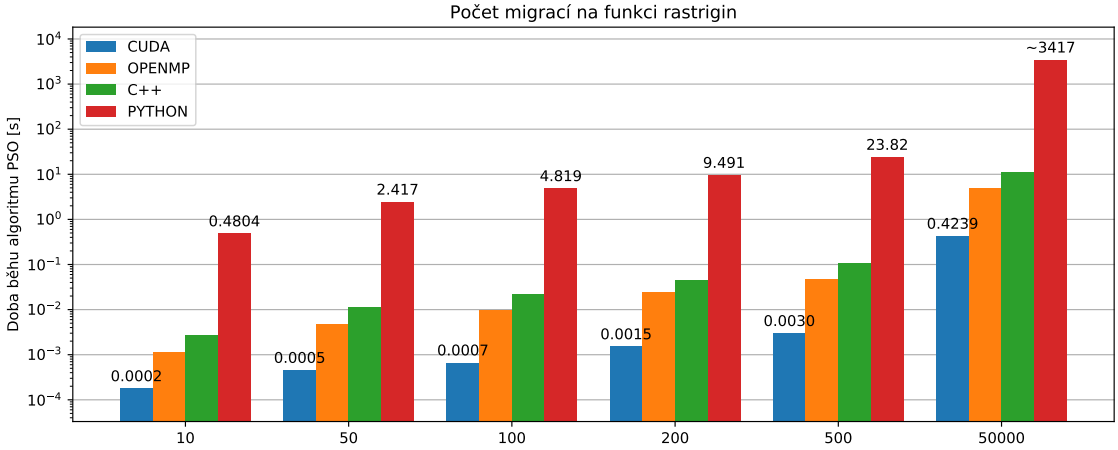
Obrázek 17: Sphere funkce



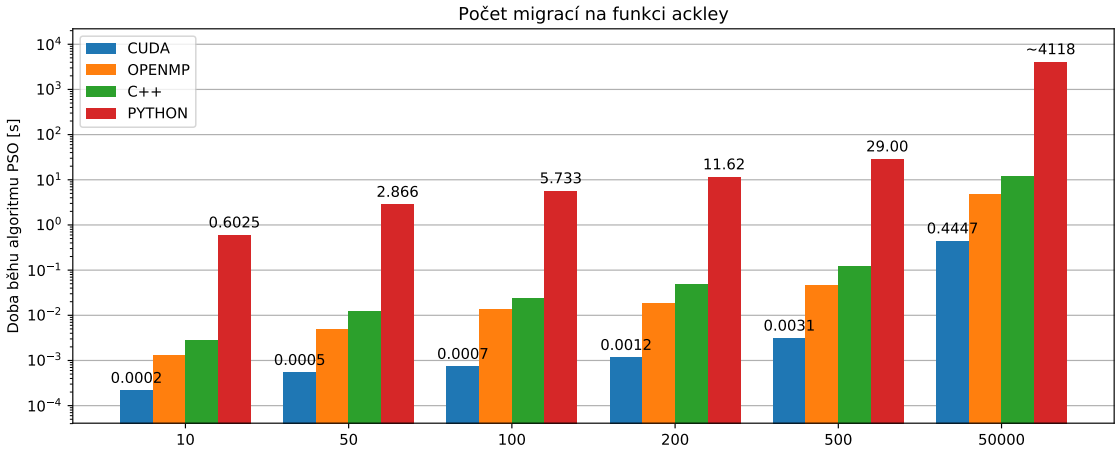
Obrázek 18: Sphere funkce s částicemi



Obrázek 19: Časy PSO na Sphere funkci



Obrázek 20: Časy PSO na Rastrigin funkci



Obrázek 21: Časy PSO na Ackley funkci

5.1.2 Rastrigin funkce

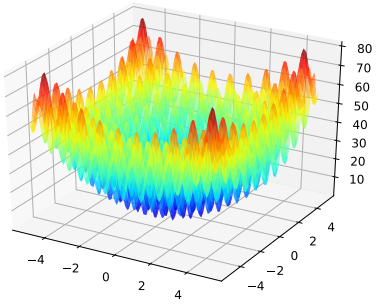
$$f(x) = 10n + \sum_{i=1}^{n} \left[x_i^2 - 10\cos(2\pi x_i) \right] \qquad (9)$$

Globální minimum:

$f(0, \ldots, 0) = 0$

Definiční obor pro hledání:

$-5.12 \leq x_i \leq 5.12$



Obrázek 22: Rastrigin funkce

5.1.3 Ackley funkce

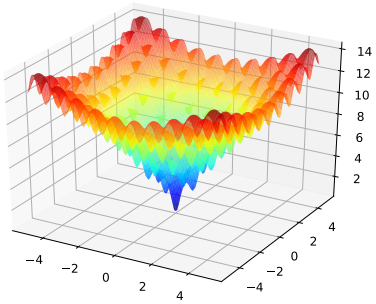
$$f(x) = -20\exp\left[-0.2\sqrt{0.5\left(\sum_{i=1}^{n} x_i^2\right)}\right] - \exp\left[0.5\left(\sum_{i=1}^{n}\cos 2\pi x_i\right)\right] + e + 20 \qquad (10)$$

Globální minimum:

$f(0, \ldots, 0) = 0$

Definiční obor pro hledání:

$-5 \leq x_i \leq 5$



Obrázek 23: Ackley funkce

5.1.4 Rosenbrock funkce

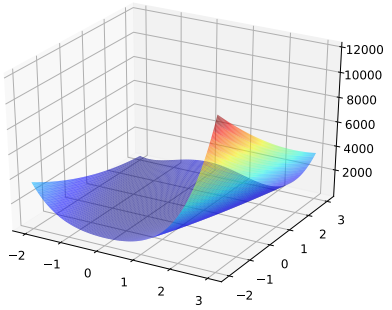
$$f(\boldsymbol{x}) = \sum_{i=1}^{n-1} \left[100\left(x_{i+1} - x_i^2\right)^2 + (1 - x_i)^2 \right] \qquad (11)$$

Globální minimum:

$f(1.0, \ldots, 1.0) = 0$

Definiční obor pro hledání:

$-2 \leq x_i \leq 3$



Obrázek 24: Rosenbrock funkce

5.1.5 Styblinski–Tang funkce

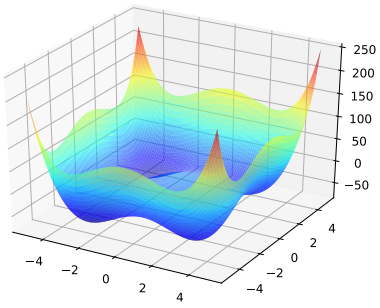
$$f(\boldsymbol{x}) = \frac{\sum_{i=1}^{n} x_i^4 - 16x_i^2 + 5x_i}{2} \qquad (12)$$

Globální minimum:

$f(-2.903534, \ldots, -2.903534) = -39.16599n$

Definiční obor pro hledání:

$-5 \leq x_i \leq 5$



Obrázek 25: Styblinski–Tang funkce

5.1.6 Schwefel funkce

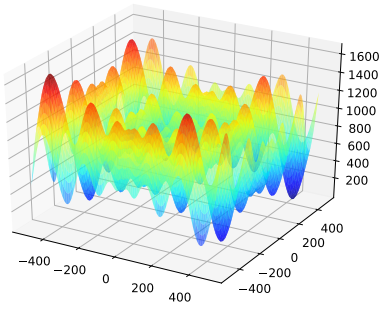
$$f(x) = 418.9829d - \sum_{i=1}^{d} x_i \sin\left(\sqrt{|x_i|}\right) \qquad (13)$$

Globální minimum:

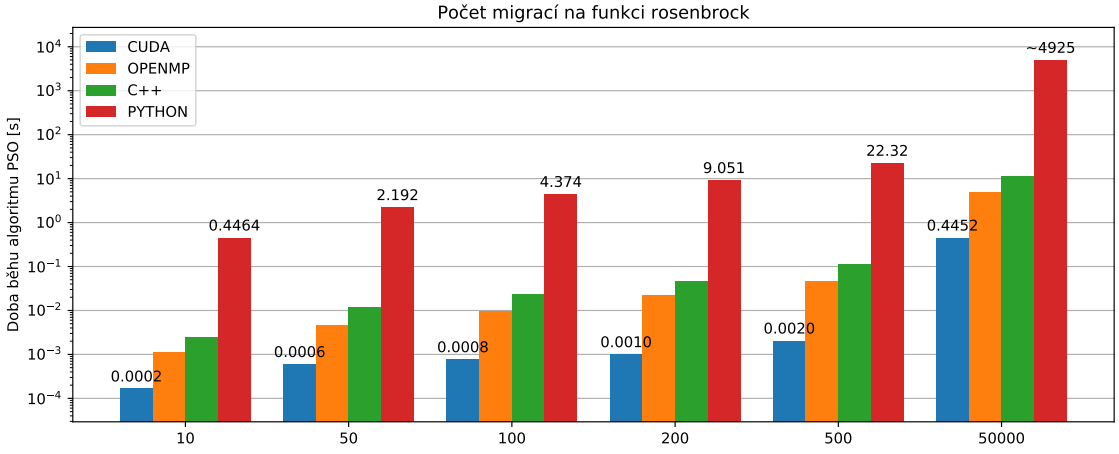
$f(420.9687, \ldots, 420.9687) = 0$

Definiční obor pro hledání:

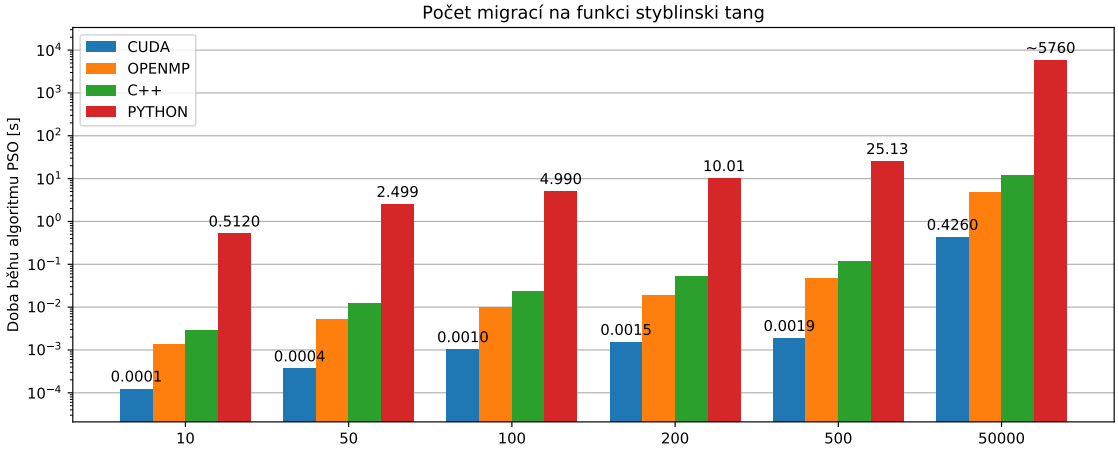
$-500 \leq x_i \leq 500$



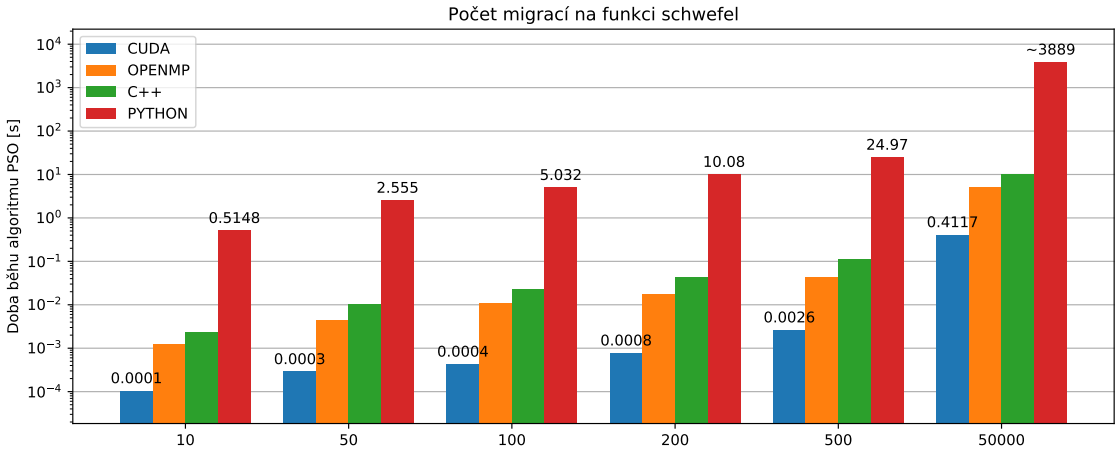
Obrázek 26: Schwefel funkce



Obrázek 27: Časy PSO na Rosenbrock funkci



Obrázek 28: Časy PSO na Styblinski–Tang funkci



Obrázek 29: Časy PSO na Schwefel funkci

5.2 Nastavení bloků a vláken pro kernel

Každý kernel na grafické kartě, je nutné spustit s parametry počet bloků a počet vláken v bloku. Toto nastavené většinou závisí na velikost dat a způsobu zpracování těchto dat, ale je zde taky hardwarový limit, například počet vláken je omezen na 1024 a v jednom bloků může být použito maximálně 65536 registrů. Byl proveden test pro zjištění nejvýkonějších parametrů pro PSO. Test byl proveden s parametry:

- $N = 1024$
- $D = 128$

Limit výpočtu byl omezen časem, a to 0,1 sekundy, do tohoto času byla započítána i alokace paměti. V tomto testu nešlo o přesnost výpočtu, ale pouze o počet vykonaných migrací za čas,

Tabulka 2: Výsledky provedeného počtu migrací PSO s různými kernel parametry

počet vláken v bloku	počet bloků										
	1	2	4	8	16	32	64	128	256	512	1024
1	3	6	14	27	61	107	229	366	544	508	581
2	6	14	29	56	120	199	396	653	833	820	780
4	12	28	59	111	230	344	691	700	1082	1069	910
8	2	54	105	218	391	607	935	1090	1152	1118	1042
16	40	92	185	332	635	881	**1264**	1168	1040	1191	1099
32	74	144	280	486	861	1171	1161	1208	1139	1204	992
64	148	271	473	797	1066	947	1095	1089	1095	992	907
128	209	387	655	1083	1068	925	1082	1092	1063	997	818
256	20	378	659	616	669	556	682	645	641	503	256
512	207	390	366	395	393	349	361	372	313	162	0
1024	203	206	188	205	204	198	170	164	94	0	0

takže ostatní parametry PSO nejsou v tomto kontextu důležité. Na konci každé migrace se zkontrolovala doba běhu, a pokud již byla delší než 0,1 sekund, algoritmus se ukončil. Nejvíce migrací se povedlo spočítat s počtem bloků nastavených na 64 a počtu vláken v bloku na 16, ostatní kombinace jsou v tabulce 2. Test byl proveden i s větší populací, ale výsledek byl skoro stejný vždy stejný, u větších populací byly lepší výsledky s parametrem počet bloků nastavený na 128. V tabulce lze také vidět, že některá nastavení nespočítaly ani jednu migraci, důvod je moc dlouhý čas, který strávily na alokaci potřebných zdrojů.

5.3 Porovnání implementací v Python, C++, OpenMP, CUDA

Na obrázcích 19, 20, 21, 27, 28, 29 je znázorněna doba výpočtu pro každou testovací funkci. Nastavení PSO bylo vždy stejné, měnil se pouze počet migrací, ostatní parametry byly:

- $N = 1024$
- $D = 2$
- $c_1 = 2, c_2 = 2$
- v_{max}= 20% rozsahu celkové plochy
- $v_{start} = 0.9$
- $v_{end} = 0.4$

Význam parametrů je popsán v algoritmu 3. Poměr časů mezi implementacemi se vyvíjel na každé funkci poměrně stejně. Při malém počtu migrací byl poměr časů nejmenší, kde CUDA provedla výpočet zhruba 2700× až 6000× rychleji než Python. Při zvyšovaní počtu migrací tento

Tabulka 3: Časy výpočtů PSO v sekundách podle migrací na Ackley funkci

	Počet migrací					
	10	50	100	200	500	50000
CUDA	0,000109	0,000297	0,000492	0,000875	0,00217	0,4118
OpenMP	0,003281	0,007531	0,014430	0,028510	0,07049	5,0686
C++	0,002917	0,012456	0,024525	0,050623	0,12318	11,861
Python	0,602514	2,865581	5,732794	11,62283	28,9959	4117

rozdíl značně rostl, kde při 500 migracích byla CUDA až 13300× rychlejší než Python verze. Pro srovnání bylo na CUDĚ spuštěno PSO s 50000 migracemi, kde čas výpočtu byl skoro stejný jako 10 migrací v Python verzi. Z výsledků s méně migracemi lze usoudit, že 50000 migrací v Pythonu by se provádělo přes 60 minut. Všechny časy výpočtů s 50000 migracemi u Python verze jsou pouze odhady na základě předešlých výsledků. Přesné časy výpočtů na Ackley funcki, kde byl rozdíl největší, jsou v tabulce 3. Na základě těchto času při 500 migracích lze vypočítat, že C++ implementace byla 235× rychlejší než Python, při použití OpenMP se výpočet v porovnání k C++ zrychlil $2,34\times$ a CUDA implementace je oproti OpenMP $12,3\times$ rychlejší. PSO tedy mělo smysl v CUDĚ implementovat.

Stejný test byl proveden pro implementace diferenciální evoluce s těmito pramatery:

- $N = 32$
- $D = 128$
- $CR = 0.6$
- $F = 0.5$

Popis těchto parametru pro DE je popsán v algoritmu 4. Poměry časů mezi Python a CUDA byly velmi podobné jako u PSO, kdy v nejlepším případě bylo dosaženo 15200násobného zrychlení. Při 500 generacích bylo mezi OpenMP a CUDA 96násobné zrychlení, ale při 50000 generacích už byla CUDA pouze $1,95\times$ rychlejší. Toto mohlo způsobit generování 3 unikátních rodičů pro každého jedince, což bylo na grafické kartě implementováno pomocí podmínek. Zrychlení mezi C++ a OpenMP rostlo s počtem generací, kdy při 500 migracích byla OpenMP implementace pouze $1,68\times$ rychlejší, ale při 50000 generacích už zrychlení bylo 3,32násobné. Všechny časy jsou vypsány v tabulce 4.

Tabulka 4: Časy výpočtů DE v sekundách podle generací na sphere funkci

	Počet generací					
	10	50	100	200	500	50000
CUDA	0,0000252	0,000115	0,000148	0,000260	0,000573	2,6791
OpenMP	0,0014547	0,005783	0,011004	0,021796	0,054074	5,2337
C++	0,0022804	0,010156	0,019055	0,038478	0,091012	17,381
Python	0,1724491	0,926951	1,720436	3,664689	8,724529	39555

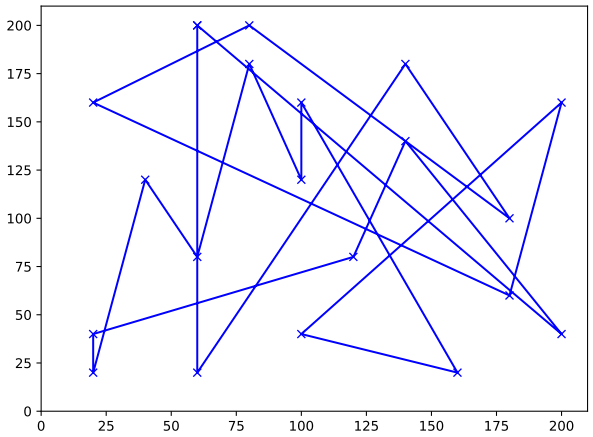
Dále byla otestována optimalizace mravenčí kolonií na problému obchodního cestujícího. Pro porovnání implementací ACO byly použity parametry:

- $\rho = 0.6$
- $\alpha = 1$
- $\beta = 1$
- počet iterací = 100

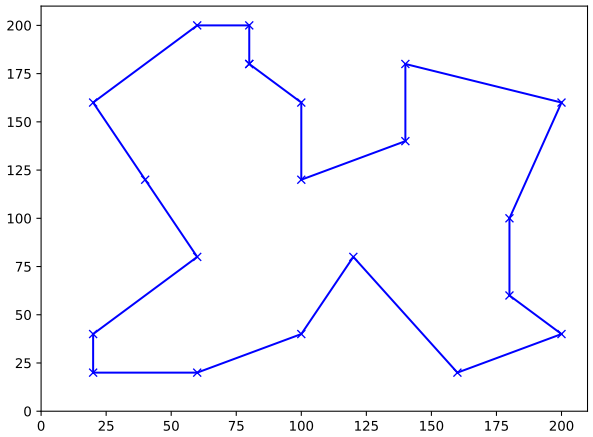
Tabulka 5: Tabulka s výsledky problému obchodního cestujícího na datasetech

Dataset	Počet měst	Doba výpočtu v ms			
		Python	C++	OpenMP	CUDA
Custom20	20	3491	6,24	5,19	40,47
Berlin52	52	55387	103,45	30,36	230,08
KroA100	100	372511	819,88	245,15	919,84

ACO bylo otestováno na datasetech KroA100, Berlin52 a Custom20 (ručně vytvořený dataset). Na datasetu Custom20 v Pythonu byl výpočet proveden za 3491 ms zatímco CUDA to zvládla za 40,47 ms. O proti PSO a DE se jedná pouze o 86násobné zrychlení. Důvod toho menšího zrychlení než u jiných EA může mít za příčinu malá velikost řešeného problému, kde se na kartě pracuje pouze s 20 vlákny. Hlavní důvod ale je implementace výběru dalšího města podle pravděpodobnosti, která se provádí pomocí podmínek, což kartu hodně zpomalilo. Tento algoritmus byl $3,7\times$ až $7,8\times$ rychlejší v C++ implementaci, viz tabulka 5. Zajímavé také je, že sériová implementace v C++ byla na malém datasetu Custom20 rychlejší než OpenMP implementace. Nejlepší řešení první a poslední iterace na datesetu Custom20 je možné vidět na obrázcích 30 a 31.



Obrázek 30: Nejlepší cesta první iterace

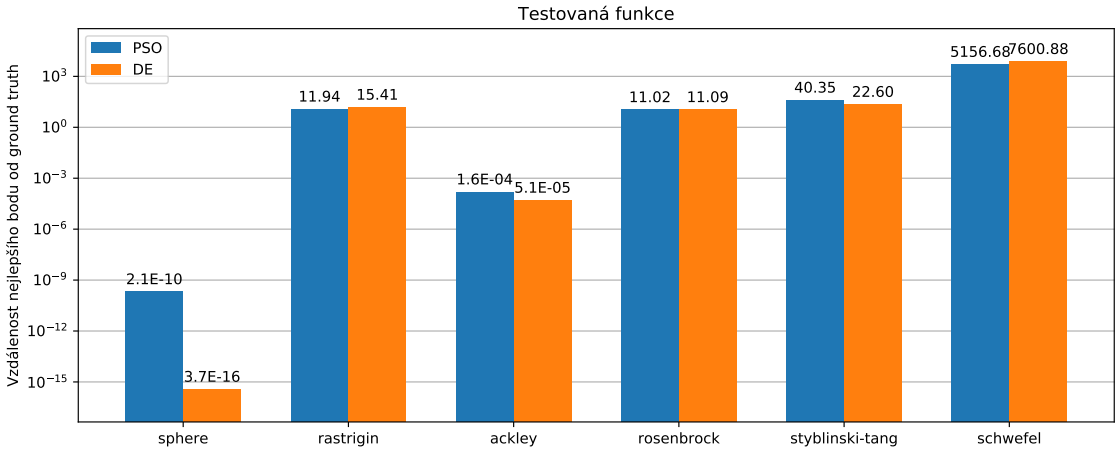


Obrázek 31: Nejlepší cesta poslední iterace

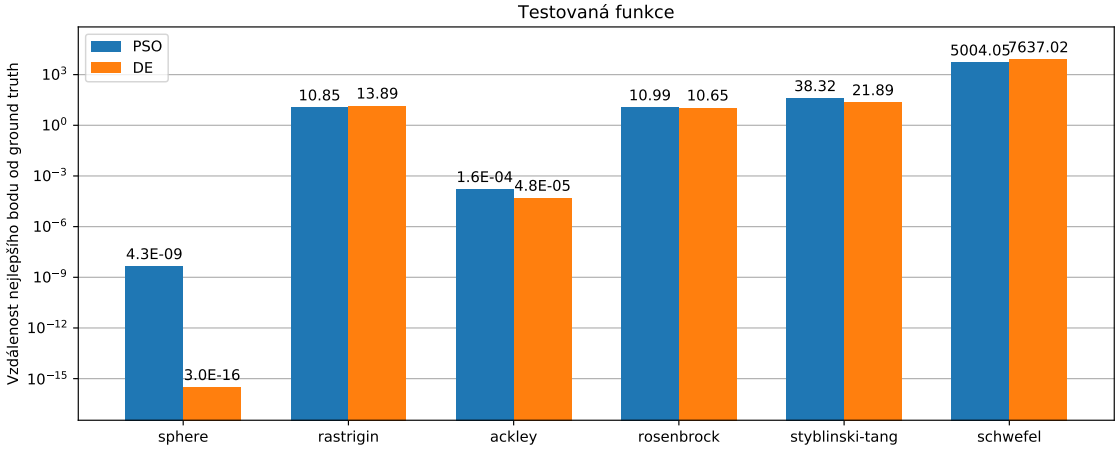
5.4 PSO vs DE

V tomto testu byly porovnány pouze implementace v CUDĚ. Byla zvolena doba, 1 sekunda, po kterou měl algoritmus čas řešit danou úlohu. Čas se začal měřit při spuštění algoritmu a kontroloval se na konci každé migrace/generace. Pokud daný čas byl překročen, nová migrace/generace už se nezačínala a ze zatím dosažených výsledků se našel ten nejlepší jedinec. Pro PSO a DE byly zvoleny stejné parametry jako v kapitole 5.3, až na dimenzi, která byla u PSO na 128. Mezi nejlepším výsledkem a skutečným globálním minimem se vypočítala Euklidovská vzdálenost, tyto výsledky jsou zobrazeny v obrázku 32. Dvě funkce kde PSO bylo lepší je funkce schwefel a rastrigin, příčina může být jejíjich rychle měnící se multimodální tvar, který PSO více vyhovuje. Výsledek na funcki rosenbrock byl u obou algoritmů téměř totožný.

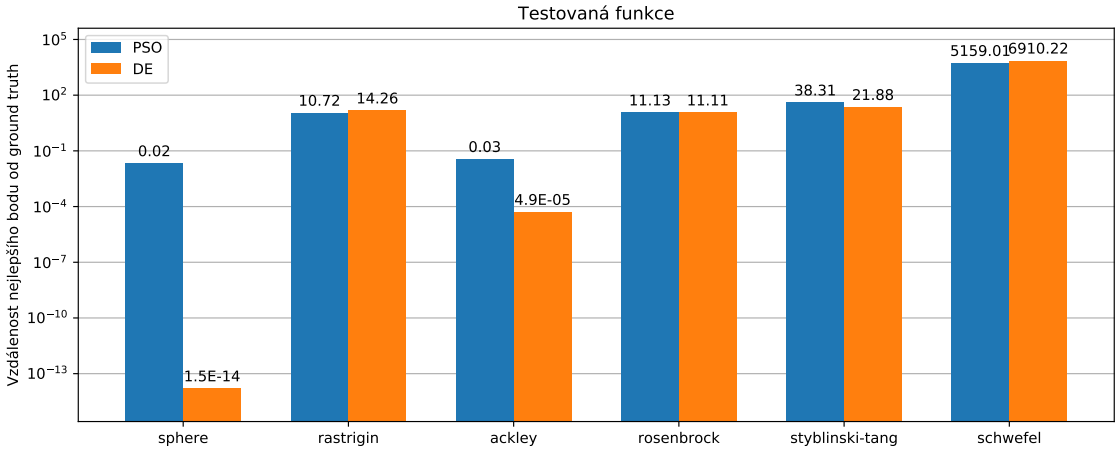
Byly také provedeny další testy, kde byla u PSO zvýšená populace na 2048 a 16384, větší populace na úkor počtu migrací ale PSO nepomohla. Na obrázcích 33 a 34 je vidět, že testy dopadly stejně. U skoro všech funkcí jsou výsledky DE lepší než PSO, kromě schewefel a rastrigin funkce, kde bylo lepší PSO. Mírně odlišené výsledky u DE jsou z důvodu náhodně generovaných čísel při výpočtu, při každém spuštění, i na stejné funkci, může dát trochu jiný výsledek.



Obrázek 32: Vzálenost výsledků od nejnižšího bodu na funkci, kde pso populace = 1024

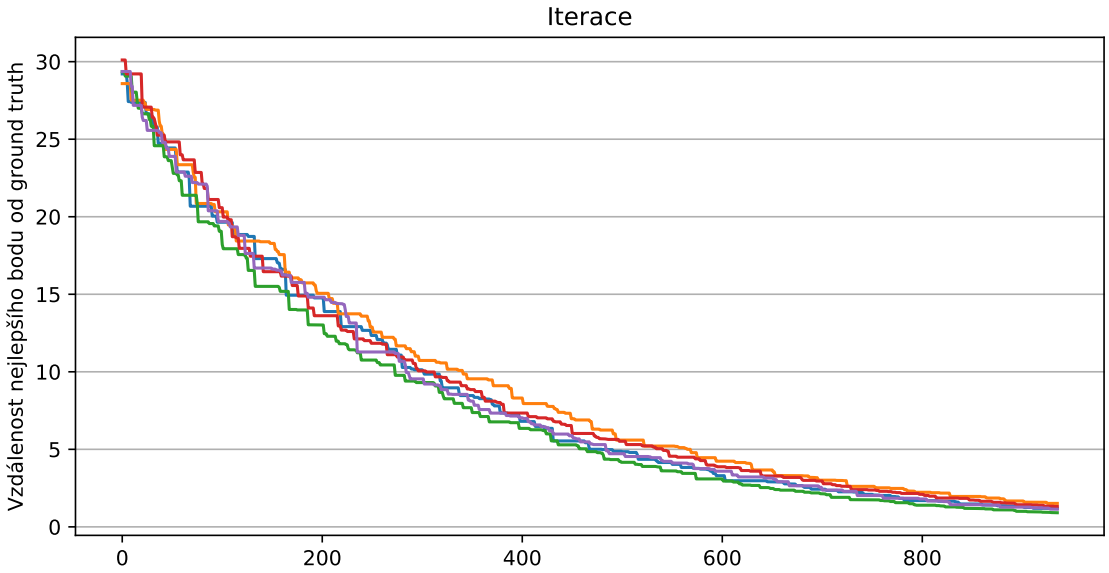


Obrázek 33: Vzálenost výsledků od nejnižšího bodu na funkci, kde pso populace = 2048

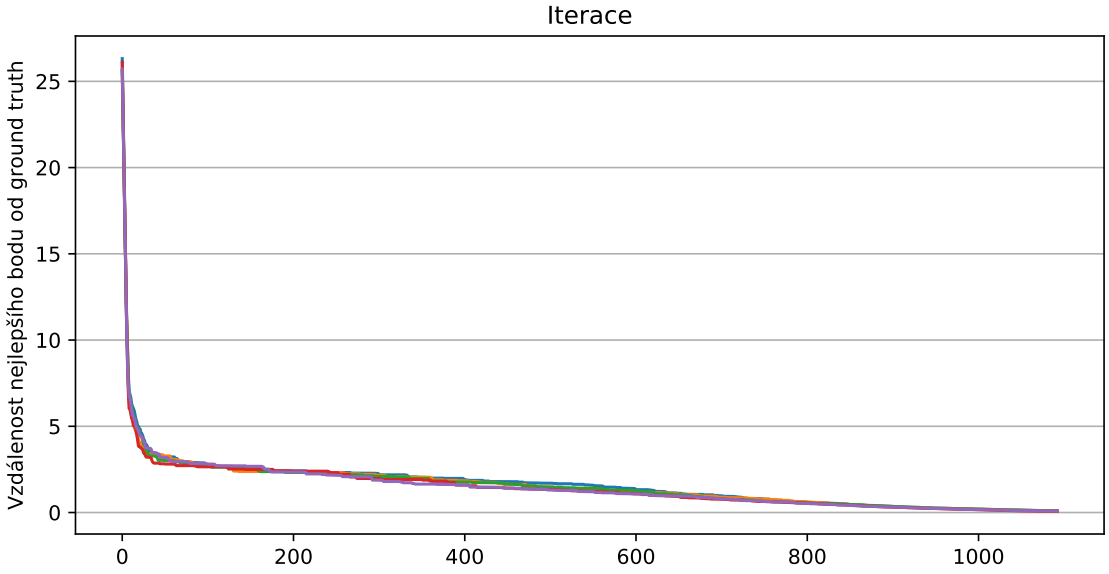


Obrázek 34: Vzálenost výsledků od nejnižšího bodu na funkci, kde pso populace = 16384

Na obrázcích 35 a 36 lze vidět vývoj nejlepšího nalezeného bodu v každé generaci/migraci. Z grafů lze vypozorovat, že PSO konvergovala k řešení mnohem rychleji než DE. Zatímco PSO se dostalo pod hranici 5 někdy okolo 30 migrace, DE se na stejnou hranici dostala až u 500 generace. Každá čára na grafu znázorňuje jeden celý výpočet, na obou grafech je znázorněno 5 výpočtů, ale u PSO si byly výsledky až moc podobné, čáry se tak překrývají.



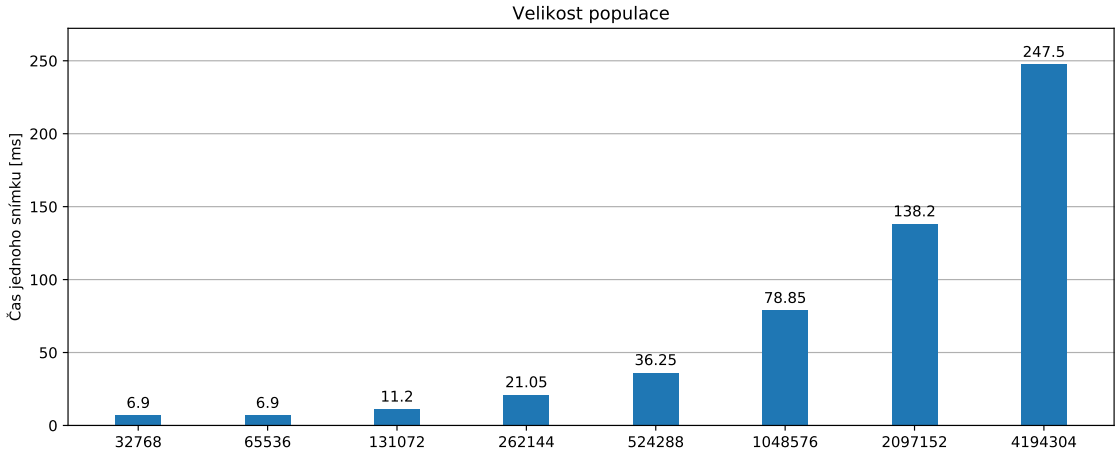
Obrázek 35: Vývoj vzálenosti výsledku DE od nejnižšího bodu na sphere funkci



Obrázek 36: Vývoj vzálenosti výsledku PSO od nejnižšího bodu na sphere funkci

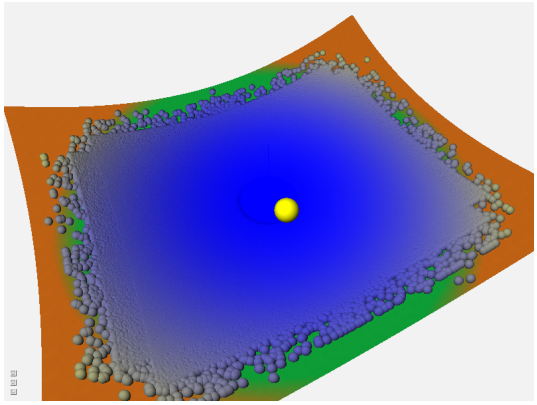
5.5 Zátěžový test vykreslování populace

Účelem toho testu bylo zjistit, jaký vliv má počet populace na rychlost snímku. V každém snímku se počítá pozice pomocí interpolace, a jednou za 20 snímků (záleží na nastavení, může být i jiné číslo) se provede opravdová migrace částic. Tento snímek je tedy o něco pomalejší, než snímky, kde se počítá pouze interpolace. Toto zpomalení je ale velmi zanedbatelné, jednalo se o zvýšení o 7 %. U populace s 4194304 prvky bylo zpoždění 20 ms a u populace 2097152 bylo zpoždění 10 ms. Z obrázku 37 je možné vyčíst, že závislost času na velikosti populace je lineární. U populací menší jak 65536 se čas již nezmenšoval, důvodem byla omezená obnovovací frekvence monitoru 144 Hz, z čehož vyplývá čas 6,9 ms na jeden snímek.



Obrázek 37: Závislost doby vykreslení jednoho snímku na velikosti populace

Velikost populace 4194304 je opravdu pouze použitelná pro zátěžové testy, s takovým počtem částic není skoro žádný pohyb částic vidět, protože částice zakrývají téměř celou hledací plochu, viz obrázek 38.



Obrázek 38: 4194304 prvků vizualizováno na sphere funkci

6 Závěr

Cílem práce bylo implementovat optimalizaci pomocí roje částic na grafické kartě v CUDA API, otestovat algoritmus a navíc vizualizovat průběh výpočtu. Aby bylo možné tento algoritmus s něčím porovnat, byl navíc implementován algoritmus diferenciální evoluce verze DE/rand/1/bin a také algoritmus optimalizace mravenčí kolonii. Všechny tyto algoritmy byly implementovány v CUDĚ, v C++ s využitím OpenMP a v jazyce Python, aby mohla být srovnána rychlost paralelní a sériové implementace.

V CUDĚ byly proti sobě porovnány algoritmy PSO a DE na 6 testovacích funkcích, kde oba algoritmy měly stejný čas na nalezení nejlepšího výsledku. Ukázalo se, že na 4 těchto funkcí byla DE lepší než PSO. Dvě funkce, na kterých byla PSO lepší, měly rychle měnící se multimodální tvar, což algoritmu DE dělalo problémy.

Dále byly mezi sebou porovnány implementace v Pythonu, C++, OpenMP a v CUDĚ. U PSO bylo nejmenší zrychlení mezi CUDA a Python implementací 3700násobné, při zvyšování počtu migrací tento rozdíl jenom přibýval, kde u 500 migrací bylo zrychlení až 13300násobné. Implementace v Pythonu již u 10 migrací trvala kolem půl sekundy, zhruba stejnou dobu trvalo 50000 migrací v CUDĚ, což za stejný čas vypočítá několikanásobně lepší výsledek. Podle odhadu by Python implementace 50000 migrací počítala přes 60 minut. U DE byl rozdíl mezi časy podobný, kde CUDA byla 15200× rychlejší než Python, ale při porovnání s C++ už tak velký rozdíl jak u PSO nebyl, kde při 50000 generacích byla CUDA pouze $1,95\times$ rychlejší. Při porovnání časů implementací ACO také k velkému zkrácení času nenastalo. Může zato implementace pro výběr dalšího města podle pravděpodobnosti, která se provádí pomocí podmínek. Na nejmenším datasetu s 20 městy byla implementace na grafické kartě pouze 86× rychlejší. Tento rozdíl ale prudce rostl na větších datasetech. Tento algoritmus byl ale nejrychleji proveden v C++ s OpenMP, kde byla tato implementace na malých problémech $7,8\times$ rychlejší a na velkých $3,7\times$ rychlejší než CUDA.

Vizualizace byla implementována v aplikaci, která snímá hloubkovou kamerou umělý písek na stole, hloubkové data zpracuje a pomocí projektoru výsledek promítne zpátky na stůl. Hloubkové pole bylo použito jako účelová funkce do algoritmu a částice byly vykresleny zpátky na stůl přes projektor. Vykreslování částic bylo provedeno pomocí OpenGL, které je schopno přistupovat k proměnným, které používala CUDA. Stačilo tedy pole s částicemi prolinkovat do OpenGL. Samotné částice byly vykresleny pomocí spritů (*sprite*), což se použilo z důvodu rychlosti. Tento způsob vykreslování zvládl vykreslit i 4 milióny částic najednou, kdy jeden čas snímku byl 250 ms, čas vykreslení byl lineárně závislý na počtu částic, u menšího počtu částic se již doba jednoho snímku nesnižovala z důvodu obnovovací frekvence monitoru.

Částice PSO se musely uměle zpomalit, protože při zobrazování každé migrace částice pouze skákaly a nešel vidět žádný pohyb. První pokus o zpomalení byl docílen limitací maximální rychlosti každé částice, tento přístup však udělal PSO nefunkční. Bylo tedy využito interpolace pozice mezi dvěma migracemi. Tento způsob již vypadal vizuálně dobře, což byl hlavní důvod

dělat vizualizaci. Zobrazuje se ale něco, co se ve skutečnosti nestalo, pozice, která se zobrazuje, totiž částice nikdy nemusela navštívit.

Z dosažených výsledků v testech tedy vyplývá, pokud máme problém, který je schopen využít velký počet vláken, má smysl implementovat řešení na grafické kartě s využitím CUDA API. Co se z čísel zjistit nedá je, že implementovat řešení v tomto prostředí zabere několikrát více času než například v Pythonu. Při programování se musí dodržovat hodně pravidel, které se v jiných jazycích berou jako samozřejmost, například omezení používání podmínek v kódu, protože karta musí vykonávat tu stejnou instrukci ve stejný čas. Porušením tohoto pravidla rychle klesá efektivita výpočtu na grafické kartě. Největším problém je, když někde v kódu nastane chyba, ať už se do kernelu pošle špatný parametr, nebo se někde na grafické kartě přistoupí mimo paměť. Chybové hlášky dávají o problému úplně minimum informací a často ukazují na jiné místo, než kde problém ve skutečnosti nastal.

Literatura

[1] J. Dréo, "Aco branches." [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/a/af/Aco_branches.svg

[2] K. Rupp, "Microprocessor trend data." [Online]. Available: https://github.com/karlrupp/microprocessor-trend-data/blob/master/42yrs/42-years-processor-trend.pdf

[3] Cburnett, "Flynn's taxonomy." [Online]. Available: https://en.wikipedia.org/wiki/Flynn%27s_taxonomy

[4] A1, "Fork–join model." [Online]. Available: https://en.wikipedia.org/wiki/Fork%E2%80%93join_model

[5] nVidia, "Cuda c++ programming guide." [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[6] R. Farber, *CUDA application design and development*. Elsevier, 2011.

[7] M. Harris, "Optimizing parallel reduction in cuda." [Online]. Available: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

[8] J. Branke, M. Orbayı, and Ş. Uyar, "The role of representations in dynamic knapsack problems," in *Workshops on Applications of Evolutionary Computation*. Springer, 2006, pp. 764–775.

[9] D. Simon, *Evolutionary optimization algorithms*. John Wiley & Sons, 2013.

[10] Y. Jin, "A comprehensive survey of fitness approximation in evolutionary computation," *Soft computing*, vol. 9, no. 1, pp. 3–12, 2005.

[11] E. Mezura-Montes and C. A. C. Coello, "Constrained optimization via multiobjective evolutionary algorithms," in *Multiobjective problem solving from nature*. Springer, 2008, pp. 53–75.

[12] K. Deb, *Multi-objective optimization using evolutionary algorithms*. John Wiley & Sons, 2001, vol. 16.

[13] D. E. Goldberg, "Genetic algorithms in search," *Optimization, and MachineLearning*, 1989.

[14] P. Rabanal, I. Rodríguez, and F. Rubio, "Using river formation dynamics to design heuristic algorithms," in *International conference on unconventional computation*. Springer, 2007, pp. 163–177.

[15] T. Back, *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[16] P. D. A. EIBEN, "What is evolutionary algorithm?" [Online]. Available: http://paulbourke.net/geometry/polygonise/

[17] K. Price, R. M. Storn, and J. A. Lampinen, *Differential evolution: a practical approach to global optimization*. Springer Science & Business Media, 2006.

[18] S. Rahnamayan, H. R. Tizhoosh, and M. M. Salama, "Opposition-based differential evolution," *IEEE Transactions on Evolutionary computation*, vol. 12, no. 1, pp. 64–79, 2008.

[19] Y. Shi and R. C. Eberhart, "Empirical study of particle swarm optimization," in *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, vol. 3, 1999, pp. 1945–1950 Vol. 3.

[20] R. Mendes, J. Kennedy, and J. Neves, "The fully informed particle swarm: simpler, maybe better," *IEEE transactions on evolutionary computation*, vol. 8, no. 3, pp. 204–210, 2004.

[21] F. Neumann and C. Witt, "Runtime analysis of a simple ant colony optimization algorithm," in *International Symposium on Algorithms and Computation*. Springer, 2006, pp. 618–627.

[22] T. Stützle, M. Dorigo *et al.*, "Aco algorithms for the traveling salesman problem," *Evolutionary algorithms in engineering and computer science*, vol. 4, pp. 163–183, 1999.

[23] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE transactions on computers*, vol. 100, no. 9, pp. 948–960, 1972.

[24] R. J. Gove, K. Balmer, N. K. Ing-Simmons, and K. M. Guttag, "Multi-processor reconfigurable in single instruction multiple data (simd) and multiple instruction multiple data (mimd) modes and method of operation," May 18 1993, uS Patent 5,212,777.

[25] P. Pacheco, *An introduction to parallel programming*. Elsevier, 2011.

[26] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[27] W. W. Fung and T. M. Aamodt, "Thread block compaction for efficient simt control flow," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 2011, pp. 25–36.

[28] B. S. Jaegeun Han, *Learn CUDA Programming: A beginner's guide to GPU programming and parallel computing with CUDA 10.x and C/C++*. Packt Publishing, 2019.

[29] D. Storti and M. Yurtoglu, *CUDA for engineers: an ction to high-performance parallel computing*. Addison-Wesley Professional, 2015.

[30] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming, portable documents*. Addison-Wesley Professional, 2010.

[31] I. ZELINKA, “Biologicky inspirované vỳpočty: evoluční algoritmy.”

[32] M. Dorigo, A. Colorni, and V. Maniezzo, “Distributed optimization by ant colonies,” 1991.

[33] H. Sarbolandi, D. Lefloch, and A. Kolb, “Kinect range sensing: Structured-light versus time-of-flight kinect,” *Computer vision and image understanding*, vol. 139, pp. 1–20, 2015.